DOCUMENTS FOR USE WITH THE UNIX TIME-SHARING SYSTEM

Hybrid PWB - V7

The enclosed UNIX documentation is supplied
in accordance with the Software Agreement
you have with the Western Electric Company.

Western Electric

Patent Licensing

Guilford Center
P. O. Box [illegible]
Greensboro, N.C. 27[illegible]
919 697 [illegible]

OCT 03 1980

TECHNISCHE HOGESCHOOL DELFT
Department of Mathematics
Juianalaan 132
2628 BL Delft
The Netherlands

Attn: Mr. P. J. van der Hoff

Gentlemen:

Re: May 1, 1979 Software Agreement Between Us
Relating to PWB/UNIX* Time Sharing Operating
System

In response to the August 22, 1980 request from Mr. van der Hoff, your institution may use the licensed software pursuant to the referenced agreement on the following additional CPU's:

LSI-11, Serial No. WM 1720
PDP 11/60, Serial No. AG 00073
Technische Hogeschool Delft
Department of Mathematics - Comp. Rm. 0.101
Julianalaan 132
2628 BL Delft
The Netherlands

Yours truly,

O. L. Wilson

O. L. WILSON
Patent Licensing Manager

*UNIX is a trademark of Bell Laboratories.

Copyright 1979, Bell Telephone Laboratories, Incorporated. Holders of a UNIX™ software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

CONTENTS

1. UNIX Introduction

2. UNIX For Beginners - Second Edition

3. A Tutorial Introduction to the UNIX Text Editor

4. An Introduction to the C Shell

5. SED - A Non-Interactive Text Editor

UNIX INTRODUCTIE

versie 3.1a december 1984

Erik van Konijnenburg
Paul Spee

Onderafdeling der Wiskunde en Informatica, TH Delft

In dit bundeltje worden de UNIX systemen geintroduceerd die draaien op de PDP 11/60 en de Geminix van de groep programmeertalen van de vakgroep informatica van de Onderafdeling der Wiskunde en Informatica van de TH-Delft. Aan bod komen:

- de punten waarin dit systeem verschilt van het originele systeem
- de programmeercyclus voor kleine en middelgrote programma's,
- een introductie in de documentatie,
- de regels voor taakwerkers
- de voorschriften voor het taakverslag.

Hoewel dit stuk is voortgekomen uit een introductie tot de PDP 11/60 bevat het ook informatie die relevant is voor taakwerkers op andere machines.

Verwijzingen in de tekst van de vorm "naam(N)" verwijzen naar de beschrijving van "naam" in hoofdstuk "N" van het UNIX programmer's Manual [UPM].

Deze bundel bevat naast deze introductie de volgende documenten:

- Unix for beginners
- A tutorial introduction to the UNIX text editor
- an Introduction to the C shell
- PWB/MM manual
- Typing documents with PWB/MM

Een tweede bundel bevat de volgende documenten:

- Programming in C - A tutorial
- C reference manual
- Unix Programming
- Tutorials voor Adb, Lex, Make, Sed en Yacc

1. Locale Informatie

1.1 Control-toetsen

De voornaamste wijzigingen behelzen de control-toetsen van de terminals; hieronder staan voor de volledigheid alle beschikbare functies gegeven met hun oude en huidige intoetswijze (^d betekent: terwijl "CTRL" ingedrukt is, "d" intoetsen).

orig. UNIX:	thd UNIX:	functie:
#	RUBOUT, soms DEL	maak laatst ingevoerde karakter ongedaan,
@	^u	maak laatst ingevoerde regel ongedaan,
DEL	^c	stop en beeindig laatst opgegeven commando,
^d	^d	geef huidig van terminal lezend commando de aanwijzing dat geen invoer meer wordt gegeven; voor de commandointerpretator is dit equivalent met uitloggen,
	^s	stopt het sturen van uitvoer naar de terminal; als een programma iets wil uitvoeren zal het blijven wachten,
	^q	de terminal kan weer uitvoer verwerken,
BREAK	^\	als ^c maar bovendien: forceer een coredump (een file "core", zie adb(I)).
	^p	zet paginering uit gedurende het huidige proces. Dit werkt alleen op de PDP 11/60.
	^w	verwijdert het laatste woord.
	^r	drukt de tot dan toe ingetikte regel opnieuw af.

1.2 Paginering

Op de *PDP*11/60 is de uitvoer naar het scherm onderverdeeld in pagina's, de uitvoer stopt automatisch na 24 regels. <return> geeft 1 regel meer,elk ander character geeft weer 24 regels. Lengte en breedte van een pagina zijn instelbaar met het stty commando of via de systemcall `ioctl'.
Zie verder de manuals *stty(1)*, *ioctl(2)* en *tty(4)*.

1.3 Printers

Aan de *PDP*11/60 is een regeldrukker verbonden. Met de commando's:

lpr [filenaam ...]
print [filenaam ...]

kunnen hierop files afgedrukt worden.

Op de Geminix stuurt *lpr*(I) files naar een (langzame) matrix printer. Het commando *opr60*(I) stuurt een file naar de 11/60 en laat het daar afdrukken.

Beide machines staan in verbinding met een letterwielprinter, de Qume. Deze printer is alleen bedoeld voor definitieve versies van verslagen e.d. Zie het manual

qume(I)

Afdrukken met het commando lpr(I) gaat als volgt: "lpr filenaam" drukt de file "filenaam" letterlijk af. Opr, gebruikt als laatste element in een 'pipe', drukt al zijn invoer letterlijk op de regeldrukker af. Zo komt vaak voor "pr filenaam | lpr &" dat de file "filenaam", netjes ingedeeld in bladzijden met kop en nummering (dit doet pr(I)), aflevert aan lpr (via de pipe "|"), die dit resultaat op de regeldrukker zet; dit geheel doet zijn werk zonder dat de gebruiker hoeft te wachten, omdat aan het eind "&" is toegevoegd. de commandos 'print', en op de Geminix 'opr60' en 'lpr' werken op een soortgelijke manier. Pipes, standaard in- en uitvoer worden verklaard in sh(I).

1.4 Fileopslag

Iedere gebruiker heeft een eigen directory. Voor taakwerkers op de PDP 11/60 is dit "/user/studi/xxx", met i = 1 of 2, waarin "xxx" de login-naam van de gebruiker is. Op welke schijf deze directory staat, is in het UNIX-systeem volkomen onbelangrijk voor de gebruiker. Een feit is wel dat de directory op een schijf staat en de ruimte daarop eindig is; gebruikers mogen daarom, als ze op een moment niet werken aan het systeem, niet teveel ruimte innemen. Dit ruimtebeslag wordt gemeten in blokken - een blok bevat 512 bytes of karakters - en kan bepaald worden met de informatie die de commando's ls(I) en du(I) geven.

Hoeveel ruimte iemand mag innemen wordt bepaald door de drukte op het systeem. Als het goed functioneren van het systeem dit vereist, kunnen er aanwijzingen gegeven worden de hoeveelheid blokken te beperken. Gebruik indien mogelijk de commando's *compact(I)* en *uncompact(I)*; hiermee kan het ruimtebeslag van een weinig gebruikt bestand tot dertig procent gereduceerd worden.

2. Programmeercyclus

Hieronder volgt een beknopte beschrijving van de cyclus die wordt doorlopen bij het programmeren van kleine en middelgrote problemen (in C) onder UNIX.

Ook voor kleinere problemen is een implementatie in de vorm van packages gewenst. De conventionele manier om packaging in C te beschrijven is als volgt:

- in een file, zeg pack.c, worden een aantal functies en variabelen gemaakt. Wanneer een object alleen binnen het file zichtbaar hoeft te zijn kan dit object 'static' worden gemaakt [C–TUT].

  Maak de packages niet te complex; een goede regel is dat geen enkel .c file langer mag zijn dan ± 10 pagina's.

- in een bijbehorend file, pack.h, worden voor objecten die buiten het package zichtbaar moeten zijn 'extern' declaraties opgenomen.

- Wanneer een ander package, user.c, functies uit pack.c nodig heeft, dan zal in user.c de volgende regel voorkomen:

  ```
  #include "pack.h"
  ```

  Hierdoor worden de juiste declaraties zichtbaar.

De compilatie van de diverse packages wordt geautomatiseerd door make(I), zie ook [MAKE]. Dit programma kijkt in een "makefile" hoe een programma gemaakt moet worden en geeft daarna de nodige opdrachten. Gebruik van make(I) verkleint niet alleen de kans op fouten, het is ook een vorm van documentatie die zeer gewenst is wanneer ook anderen uw werk moeten gebruiken.

1. De probleemuitwerking geschiedt zonder computerhulp: hierbij kunnen de taal- en subroutinebeschrijvingen steun verlenen. Vooral de "Standard IO library", beschreven in [U–PROG] bevat vele nuttige en krachtige routines. Bij programmeren in C is het gebruik hiervan zeer aanbevolen.

Andere automatisch beschikbare routines staan beschreven in hoofdstuk twee en drie van [UPM].

Zie "C Tutorial" [C–TUT], het "C Reference Manual" [C–REF] en "Unix programming" [U–PROG].

2. Intypen van een nieuw programma of het wijzigen van een bestaand programma met naam "filename":

   em filename

   em (I) is een uitbreiding van de editor ed (I). van em (I) bestaan alleen de manual pages, ed (I) is ook beschreven in: "UNIX For Beginners" [U–TUT] en "A Tutorial Introduction to the UNIX Text Editor" [ED–TUT].

3. Het compileren van een C programma; de filenaam moet eindigen op ".c":

   cc [-o output–file] input–file

   Wanneer geen output file wordt opgegeven zal cc zelf een naam kiezen; voor complete programma's is dit *a.out*. Voor naam.c, dat nog met andere files gelinkt moet worden, zal de compiler output naam.o heten.

   Als bij de compilatie fouten worden ontdekt, moet de editor weer worden gebruikt om het programma te corrigeren. Meestal zijn alleen de eerste foutmeldingen bruikbaar en zijn de rest volgfouten; vooraan de regel staat het regelnummer waarin de fout is ontdekt.

4. Het draaien van een gecompileerd programma: als de naam "xx" luidt, geef dan als commando (alle standaard commando's zijn in feite gewone programma's):

   xx

   eventueel gevolgd door argumenten als het programma dat vereist (zie de "C Tutorial" [C–TUT] bij "Program Arguments" in paragraaf 18.).

   Bij het optreden van onduidelijke fouten kan het volgende gedaan worden:

   a. extra afdrukopdrachten toevoegen aan het programma, om zodoende de loop van het programma te kunnen volgen (zie printf(III))

      In het algemeen wordt de output van een programma gebufferd. Bij een testprogramma is dit niet wenselijk, omdat het programma plat kan gaan vóordat de buffer vol is. De debugging informatie wordt dan niet afgedrukt. De routine *setbuf(III)* kan de buffering uitschakelen.

   b. een "coredump" van het programma met de debugger adb(I) bekijken; deze dump wordt gegenereerd als het programma vastloopt, of geforceerd door de gebruiker via een ^\ (zie 2.1.):

      adb xx

      Hiermee start de debugger met als programma "xx" en als coredump de gecreeerde file "core". Het commando "$c" laat zien welke routines aktief waren toen het programma gestopt werd; verder kunnen de waarden van variabelen bekeken worden (zie adb(I) en [ADB]).

3. Documentatie

De documentatie van het standaard PWB/UNIX-systeem beslaat meerdere (dikke) boeken:

- UNIX commando's [UPM] en
- UNIX documenten.

3.1 UNIX programmer's manual

Het UNIX programmer's manual bevat na een inleiding over het gebruik van het boek en indices, acht hoofdstukken die genummerd zijn met romeinse cijfers. In elk hoofdstuk staan de onderwerpen alfabetisch gerangschikt.

In I staan de commando's die van een terminal of door commandotaalprocedures (shellprocedures, zie csh(I), sh(I)) kunnen worden gegeven. In II staan de "systemcalls" vanuit UNIX assemblercode of vanuit C, waarmee vanuit programma's direkt diensten van het UNIX-OS gevraagd kunnen worden. In hoofdstuk III staan de beschikbare standaard-subroutines.

De volgende hoofdstukken zijn in het dagelijks gebruik van minder belang. Deze hoofdstukken IV t/m VIII bevatten resp. de karakteristieken van verschillende randapparaten, formaten van bijzondere files, beschrijvingen van niet-standaard programma's en subroutines en de commando's van de systeembeheerder.

De volgende commando's uit I worden veel gebruikt voor het creeren, editen, een andere naam geven en verwijderen van files

cat(I)	voeg files aan elkaar en laat ze zien (conCATenate)
cd(I)	Change Directory
cp(I)	copieer een file (CoPy)
csh(I)	een ingewikkelder versie van de gewone shell
du(I)	geef het schijfgebruik in blokken (Disk Usage)
em(I)	text editor (Editor for Mortals)
ls(I)	laat een directory zien (LiSt directory)
make(I)	automatiseer programma regeneratie (MAKE)
mkdir(I)	maak een nieuwe directory (MaKe Directory)
mv(I)	geef een file een andere naam (MoVe or rename)
pr(I)	druk files af met kop en nummering (PRint), vaak in combinatie met printer commandos, zie het betreffende hoofdstuk.
rm(I)	verwijder een file (ReMove)
rmdir(I)	verwijder een directory (ReMove DIRectory)
sh(I)	commandointerpretator (SHell)
tp(I)	manipuleer floppy disk (TaPe)

Met textprocessing hebben de volgende commando's te maken:

nroff(I)	nroff formatter (Normal Run OFF?)
tbl(I)	preprocessor voor tabellen (TaBLe)
mm(I)	een pakket macro's om het gebruik van NROFF te vereenvoudigen.

Verder zijn er nog commando's die de inhoud van files interpreteren:

comm(I)	zoek naar COMMon lines in verschillende files
csplit(I)	splits een file in stukken (Context SPLIT)
diff(I)	geef de verschillen tussen twee files (DIFFerences)
file(I)	bepaal het filetype (FILE)

grep(I)	zoek een patroon in een file (G/Regular Expression/P)
od(I)	dump een file (Octal Dump)
sed(I)	Stream EDitor, Voor editwerk met veel herhalende opdrachten
sort(I)	sorteer een file (SORT)

De volgende commando's zijn nuttig in verband met C

adb(I)	A DeBugger
as(I)	ASsembler
cb(I)	C Beautifier
cc(I)	C Compiler
ld(I)	Link eDitor (vaak ook LoaDer genoemd)
lex(I)	(LEXical analiser)
nm(I)	druk symboltable af van programma (NaMes, niet op de Geminix)
yacc(I)	(Yet Another Compiler Compiler)

3.2 UNIX documenten

De UNIX documenten zijn losse artikelen over speciale losstaande onderwerpen. Aanbevolen voor de nieuwe gebruiker:

[U–TUT]	"Unix for beginners", een algemene inleiding;
[ED–TUT]	"A tutorial introduction to the UNIX text editor", over de editor ed(I). Lees ook het manual em(I) over de geboden uitbreidingen;
[MM–TUT]	"Typing documents with PWB/MM", een pakket dat de opmaak van verslagen verzorgt;

Voor wie in C moet programmeren:

[C–TUT]	"Programming in C - a Tutorial", een informele inleiding tot de taal C;
[U–PROG]	"Unix Programming", Beschrijft de manieren waarop C programmas met de rest van het systeem kunnen omgaan. De appendix geeft een kort overzicht van de standard IO library;
[MAKE]	Geautomatiseerde compilatie

Het boek "The C Programming Language" [C] bevat naast een exemplaar van [C–REF] een zeer uitgebreide inleiding in de taal C; dit boek is aanwezig in de Onderafdelingsbibliotheek (Md 403).

4. Regels voor taakwerkers

Voor het verrichten van een taak of scriptie bij de leerstoel programmeertalen van de vakgroep informatica gelden de volgende regels.

4.1 Inschrijven voor een taakopdracht

Voor het verkrijgen van een taakopdracht kunt u bij dhr. Ververs (kamer 2.008) terecht. Hij zal u een begeleider toewijzen. De begeleider reikt u een taak uit, waarna u zich dient te laten registreren bij dhr. Pronk (kamer 2.124).

4.2 Afschrijven

Meestal is de drukte op het systeem niet zodanig dat afschrijven nodig is. Mocht het nodig worden om met lijsten te gaan werken, dan zullen de volgende regels gaan gelden:

Op terminals in kamer 0.102 kan worden afgeschreven, echter van te voren maximaal een uur per taak. Dit betekent dat nooit meer dan een uur vooruit mag worden afgeschreven, dat nooit twee mensen die aan een taak werken tegelijk kunnen afschrijven en dat als de afschrijftermijn nog loopt, niet nogmaals mag worden afgeschreven; echter een terminal die vrij is, mag altijd gebruikt worden, eventueel tot de rechthebbende komt opdagen; het afschrijfboek ligt in kamer 0.102. Taakwerkers en afstudeerders maken gebruik van dezelfde lijst.

De LA36 (tty8) in de computerruimte mag worden gebruikt voor het manipuleren van floppy disks;

4.3 Assistentie

De assistenten zijn te vinden op kamer 0.105. Onder assistentie wordt verstaan:

- Inschrijven en introductie op het systeem,
- het geven van aanwijzingen bij onbekendheid met het systeem,
- het inspringen bij een niet juist functioneren van het systeem,

en niet:

- het zoeken van fouten in programma's,
- het uitleggen van zaken die al ergens beschreven zijn.

De PDP 11/60 wordt beheerd door A. Biegstraaten, de Geminix door D. Kamstra. Beiden zijn te vinden op kamer 2.126.

4.4 Onderbreking

Voor het UNIX systeem geldt dat een onderbreking van de aan de taak te verrichten werkzaamheden van langer dan twee maanden aan de taakbegeleider *en* aan de systeembeheerder moet worden gemeld. Dit om te voorkomen dat een gebruiker zijn login-naam kwijtraakt (en dus ook zijn files).

4.5 Afmelding van een taakopdracht

Bij het afmelden van de taak dienen de volgende zaken te worden ingeleverd bij de taakbegeleider:

1. een verslag (zie het volgende hoofdstuk),
2. een floppy-disk (verstrekt door de assistent) met
   a. een bestand met de naam READ–ME, met daarin wat het programma is en doet, of het uberhaupt werkt, wie het wanneer gemaakt heeft, en voor wie,
   b. een executable versie van het gemaakte programma,
   c. alle (rijkelijk becommentarieerde) "sources" van de gemaakte programmatuur,
   d. een makefile (zie make(I)) waarmee uit de sources de executable versie kan worden geconstrueerd,
   e. het verslag in "nroff -mm" formaat,
   f. wanneer het nieuwe programma een utility is, bestemd om eventueel in de rest van het systeem geintegreerd te worden, *manuals* in het formaat dat in man(7) beschreven wordt,
   g. eventueel nader noodzakelijke of gewenste files, zoals testinvoer,

   maar zonder oude rommel of troep die niet ter zake doet,
3. alle geleende handboeken.

U dient zich te laten uitschrijven bij dhr. Pronk (kamer 2.124).

5. Taakverslag

Verslagen voor taken en scripties dienen te worden gemaakt met behulp van de onder UNIX beschikbare software. Hiervoor is afdrukapparatuur aanwezig.
Er wordt geexperimenteerd met een Qume Daisywheel printer; het is soms mogelijk om de eindversie van een verslag hierop af te drukken, dit in overleg met de systeembeheerder of assistent.

Een verslag dient tenminste de volgende punten te bevatten:

1. een titel pagina met daarop vermeld:
   a. naam van de taakwerker(s),
   b. titel van het onderzoek,
   c. datum van inlevering van de taak,
   d. datum van begin van de taak,
2. duidelijke omschrijving van de taakstelling,
3. analyse en bespreking van mogelijke alternatieve oplossingen en een gemotiveerde reden voor de keuze van de oplossingsmethode (theorie),
4. gedetailleerde bespreking van de oplossingsmethode, programmadocumentatie,
5. EEN GEBRUIKSAANWIJZING, login-naam en password,
6. resultaten en voorbeelden,
7. lijst van geraadpleegde literatuur.

Literatuur

[ADB] J. F. Maranzano, S. R. Bourne, *A Titorial introduction to ADB*, Bell Labs. In [U–DOC].

[C] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632.

[C–REF] D.M. Ritchie, *C Reference Manual*, Bell Labs, mei 1977

[C–TUT] B.W. Kernighan, *Programming in C - A Tutorial*, Bell Labs, mei 1974

[CSH–TUT] William Joy, *an Introduction to the C shell*, Berkeley

[ED–TUT] B.W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor*, Bell Labs, oktober 1974

[MAKE] S.I. Feldman, *Make - A Program for Maintaining Computer Programs*, Bell Labs, april 1977

[MM–REF] D.W. Smith and J.R. Mashey, *PWB/MM - PWB's Memorandum Macro's*, Bell Labs, oktober 1977

[MM–TUT] D.W. Smith and E.M. Piskorik, *Typing Documents with PWB/MM*, Bell Labs, oktober 1977

[NEW–IO] D.M. Ritchie, *A New Input-Output Package*, Bell Labs, juli 1977

[NROFF] J.F. Ossanna, *NROFF/TROFF User's Manual*, Bell Labs, mei 1977

[SED] L. E. Mc Mahon, *Sed - A Non-interactive Text Editor*, Bell Labs, Augustus 1978

[SH–TUT] J.R. Mashey, *PWB/UNIX Shell Tutorial*, Bell Labs, september 1977

[TBL] M.E. Lesk, *Tbl - A Program to Format Tables*, Bell Labs, september 1977

[U–TUT] B.W. Kernighan, *UNIX For Beginners*, Bell Labs, oktober 1974

[U–PROG] B.W. Kernighan and D.M. Ritchie, *UNIX Programming - Second Edition*, Bell Labs, oktober 1978

[UPM] T.A. Dolotta, R.C. Haight and E.M. Piskorik eds., *UNIX Programmers Manual - 7th edition*, Bell Labs, mei 1979

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help new users get started on the UNIX† operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated UNIX bibliography.

September 30, 1978

†UNIX is a Trademark of Bell Laboratories.

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *The UNIX Programmer's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor*, which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.
5. A UNIX Reading List. An annotated bibliography of documents that new users should be aware of.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300's; Execuport, TI and similar portables; video (CRT) terminals like the HP2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case*. If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type "**login:**" at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the "break" or

"interrupt" key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a **login:** message, type your login name *in lower case*. Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a "prompt character," a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign **$** or a percent sign %. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

date

followed by RETURN. You should get back something like

Mon Jan 16 14:17:10 EST 1978

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

```
mb    tty01   Jan 16   09:11
ski   tty05   Jan 16   09:33
gam   tty11   Jan 16   13:07
```

The time is when the user logged in; "ttyxx" is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

whom

you will be told

whom: not found

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed or a return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in section I of the manual. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type the command

stty −tabs

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character # erases the last character typed; in fact successive uses of # erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as **date**.

The at-sign @ erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an @ and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either # or @ by a backslash \, it loses its erase meaning. So to enter a sharp or at-sign in something, type \# or \@. The system will always echo a newline at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in \##. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on. It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first. After each message, **mail** waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to "joe" (assuming "joe" is someone's login name). The easiest way is this:

mail joe
now type in the text of the letter
on as many lines as you like ...
After the last line of the letter
type the character "control−d",
that is, hold down "control" and type
a letter "d".

And that's it. The "control-d" sequence, often called "EOF" for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *UNIX Programmer's Manual.*)

Writing to other users

At some point, out of the blue will come a message like

Message from joe tty07...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

> Joe types **write smith** and waits.
> Smith types **write joe** and waits.
> Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for "over".
> Now Smith types a reply, also terminated by **(o)**.
> This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for "over and out".
> To terminate the conversation, each side must type a "control-d" character alone on a line. ("Delete" also works.) When the other person types his "control-d", you will get the message **EOF** on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type "control-d".

On-line Manual

The *UNIX Programmer's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the **who** command, type

man who

and, of course,

man man

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, docum nt preparation, and even C programming. Try typing the command

learn

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX "text editor" **ed**. Since **ed** is thoroughly documented in **ed**(1) and explained in *A Tutorial Introduction to the UNIX Text Editor,* we won't spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file called **junk** with some text in it, do the following:

ed junk	(invokes the text editor)
a	(command to "ed", to add text)
now type in	
whatever text you want ...	
	(signals the end of adding text)

The "." that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

w

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you hang up and go home the information is lost.† But after **w** the information is there permanently; you can re-access it any time by typing

ed junk

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a ? to remind you. A second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

What files are out there?

The **ls** (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If you type

ls

the response will be

junk
temp

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

ls −t

causes the files to be listed in the order in which they were last changed, most recent first. The **−l** option gives a "long" listing:

ls −l

will produce something like

```
-rw-rw-rw-  1 bwk   41 Jul 22 2:56 junk
-rw-rw-rw-  1 bwk   78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). **bwk** is the owner of the file, that is, the person who created it. The **−rw−rw−rw−** tells who has permission to read and write the file, in this case everyone.

† This is not strictly true — if you hang up while editing, the data you were working on is saved in a file called **ed.hup**, which you can continue with at your next session.

Options can be combined: **ls −lt** gives the same thing as **ls −l**, but sorted into time order. You can also name the files you're interested in, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign, like **−t** and **−lt**, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l**.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in **junk** and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat**, the simplest of all the printing programs. **cat** simply prints on the terminal the contents of all the files named in a list. Thus

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two. The files are simply concatenated (hence the name "**cat**") onto the terminal.

pr produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print **junk** neatly, then skip to the top of a new page and print **temp** neatly.

pr can also produce multi-column output:

```
pr −3 junk
```

prints **junk** in 3-column format. You can use any reasonable number in place of "3" and **pr** will do its best. **pr** has other capabilities as well; see **pr**(1).

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an **ls** command now, you will get

```
precious
temp
```

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

```
cp precious temp1
```

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

```
rm temp temp1
```

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive.

Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the **ls** command, **ls −t** means to list in time order. So if you had a file whose name was **−t**, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

chap1
chap2
etc...

Or, if each chapter were broken into several files, you might have

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

pr chap1.1 chap1.2 chap1.3 ......

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

pr chap*

The * means "anything at all," so this translates into "print all files whose names begin with **chap**", listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the "shell," **sh**(1)). Using that fact, you can see how to list the names of the files in the book:

ls chap*

produces

chap1.1
chap1.2
chap1.3
...

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

rm *junk* *temp*

removes all files that contain **junk** or **temp** as any part of their name. As a special case, * by itself matches every filename, so

pr *

prints all your files (alphabetical order), and

rm *

removes *all files*. (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

pr chap[12349]*

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

pr chap[1−49]*

Letters can also be used within brackets: **[a−z]** matches any character in the range **a** through **z**.

The ? pattern matches any single character, so

ls ?

lists all files which have single-character names, and

ls −l chap?.1

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

ls '?'

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tools is the command **pwd** ("print working directory"), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

/usr/your-name

This says that you are currently in the directory **your-name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just /. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding changes and read on.)

If you now type

ls /usr/your-name

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

ls /usr

This should print a long series of names, among which is your own login name **your-name**. On many systems, **usr** is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

ls /

You should get a response something like this (although again the details may be different):

bin
dev
etc
lib
tmp
usr

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

Now try

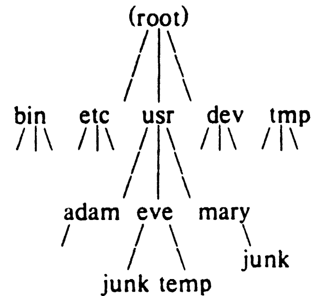
cat /usr/your-name/junk

(if **junk** is still around in your directory). The name

/usr/your-name/junk

is called the **pathname** of the file that you normally think of as "junk". "Pathname" has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:



Notice that Mary's **junk** is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

pr /usr/your-name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor-name

or make your own copy of one of his files by

cp /usr/your-neighbor/his-file yourfile

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be

arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with pathnames, try

ls /bin /usr/bin

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

cd /usr/your-friend

(On some systems, **cd** is spelled **chdir.**) Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

pwd

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

mkdir book

then go to it with

cd book

then start typing chapters. The book is now found in (presumably)

/usr/your-name/book

To remove the directory **book**, type

rm book/*
rmdir book

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

cd ..

"**..**" is the name of the parent of whatever directory you are currently in. For completeness, "**.**" is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

ls

makes a list of files on your terminal. But if you say

ls >filelist

a list of your files will be placed in the file **filelist** (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means "put the output on the following file, rather than on the terminal." Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

cat f1 f2 f3 >temp

The symbol >> operates very much like > does, except that it means "add to the end of." That is,

cat f1 f2 f3 >>temp

means to concatenate **f1**, **f2** and **f3** to the end of whatever is already in **temp**, instead of overwriting the existing contents. As with >, if **temp** doesn't exist, it will be created for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called **script**. Then you can run the script on a file by saying

ed file <script

As another example, you can use **ed** to prepare a letter in file **let**, then send it to several people with

mail adam eve mary joe <let

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

pr f g h

will print the files **f**, **g**, and **h**, beginning each on a new page. Suppose you want them run together instead. You could say

cat f g h >temp
pr <temp
rm temp

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

cat f g h | pr

The vertical bar | means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

ls | pr −3

prints a list of your files in three columns. The program **wc** counts the number of lines, words and characters in its input, and as we saw earlier, **who** prints a list of currently-logged on people, one per line. Thus

who | wc

tells how many people are logged on. And of course

ls | wc

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. **pr** is one example:

pr −3 a b c

prints files **a**, **b** and **c** in order in three columns. But in

cat a b c | pr −3

pr prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious "shell," which is in fact **sh**(1). The shell is the program that interprets what you type as commands and arguments. It also looks after translating *, etc., into lists of filenames, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

date; who

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

ed file <script &

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to say

ed file <script >script.out &

which saves the output lines in a file called **script.out**.

When you initiate a command with **&**, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

kill process-number

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you're curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

(command-1; command-2; command-3) &

to start three commands in the background, or you can start a background pipeline with

command-1 | command-2 &

Just as you can tell the editor or some simi-

lar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

sh startup

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

chmod +x startup

and thereafter you need only say

startup

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We'll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced "tee-roff") instead drives a phototypesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the "manuscript" package known as **−ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **−ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **−ms**, use the command

troff −ms files ...

for the typesetter, and

nroff −ms files ...

for a terminal. The **−ms** argument tells **troff** and **nroff** to use the manuscript package of formatting requests.

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

sum from i=0 to n x sub i ˜=˜ pi over 2

produces the output

$$\sum_{i=0}^{n} x_i = \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.

spell and **typo** detect possible spelling mistakes in a document. **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are "unusual", and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

grep 'ing$' chap*

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like * or $ that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

tr A−Z a−z <input >output

sort sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like **−ms** is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like **.PP**, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** and **troff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing commands and request definitions.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...     collect the files
|tr ...     put each word on a new line
|tr ...     delete punctuation, etc.
|sort       into dictionary order
|uniq       discard duplicates
|comm       print words in text
              but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable **i** to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else**, **while**, **for**, **case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in *An Introduction to the UNIX Shell*, by S. R. Bourne.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is also a easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently

includes at least Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX 11/780, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. **lint** checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) **make** allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **adb** is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the **−p** option; after the test run, use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **adb**, **prof**, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly. The **lex** lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a **yacc**-based program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. UNIX READING LIST

General:

K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978. Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only need to read section 1.

Documents for Use with the UNIX Time-sharing System. Volume 2 of the Programmer's Manual. This contains more extensive descriptions of major commands, and tutorials and reference manuals. All of the papers listed below are in it, as are descriptions of most of the programs mentioned above.

D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System," CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978, contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation:

B. W. Kernighan, "A Tutorial Introduction to the UNIX Text Editor" and "Advanced Editing on UNIX," Bell Laboratories, 1978. Beginners need the introduction; the advanced material will help you get the most out of the editor.

M. E. Lesk, "Typing Documents on UNIX," Bell Laboratories, 1978. Describes the **−ms** macro package, which isolates the novice from the vagaries of **nroff** and **troff**, and takes care of

most formatting situations. If this specific package isn't available on your system, something similar probably is. The most likely alternative is the PWB/UNIX macro package **−mm**; see your local guru if you use PWB/UNIX.

B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Bell Laboratories Computing Science Tech. Rep. 17.

M. E. Lesk, "Tbl — A Program to Format Tables," Bell Laboratories CSTR 49, 1976.

J. F. Ossanna, Jr., "NROFF/TROFF User's Manual," Bell Laboratories CSTR 54, 1976. **troff** is the basic formatter used by **−ms**, **eqn** and **tbl**. The reference manual is indispensable if you are going to write or maintain these or similar programs. But start with:

B. W. Kernighan, "A TROFF Tutorial," Bell Laboratories, 1976. An attempt to unravel the intricacies of **troff**.

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

B. W. Kernighan and D. M. Ritchie, "UNIX Programming," Bell Laboratories, 1978. Describes how to interface with the system from C programs: I/O calls, signals, processes.

S. R. Bourne, "An Introduction to the UNIX Shell," Bell Laboratories, 1978. An introduction and reference manual for the Version 7 shell. Mandatory reading if you intend to make effective use of the programming power of this shell.

S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Bell Laboratories CSTR 32, 1978.

M. E. Lesk, "Lex — A Lexical Analyzer Generator," Bell Laboratories CSTR 39, 1975.

S. C. Johnson, "Lint, a C Program Checker," Bell Laboratories CSTR 65, 1977.

S. I. Feldman, "MAKE — A Program for Maintaining Computer Programs," Bell Laboratories CSTR 57, 1977.

J. F. Maranzano and S. R. Bourne, "A Tutorial Introduction to ADB," Bell Laboratories CSTR 62, 1977. An introduction to a powerful but complex debugging tool.

S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler," Bell Laboratories, 1978. A full Fortran 77 for UNIX systems.

A Tutorial Introduction to the UNIX Text Editor with some Notes on EM

B. W. Kernighan

Bell Laboratories, Murray Hill, N. J.

ABSTRACT

In the past, allmost all text input on the UNIX operating system was done with the text editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

Furthermore, some notes on the use of *em*, a superset of *ed* are given. This editor offers an "open mode" that makes it easier to make small corrections within a line.

December 12, 1984

A Tutorial Introduction to the UNIX Text Editor with some Notes on EM

B. W. Kernighan

Bell Laboratories, Murray Hill, N. J.

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the UNIX manual, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.)

This text also contains some notes on *em*, a superset of ed. Em has an "open mode", simplifying small changes in existing text. When working on the PDP 11/60, you will probably use *em*.

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is.

You must also know what character to type as the end-of-line on your particular terminal. This is a "newline" on Model 37 Teletypes, and "return" on most others. Throughout, we will refer to this character, whatever it is, as "newline".

Getting Started

We'll assume that you have logged in to UNIX and it has just said "%". The easiest way to get *ed* is to type

```
ed    (followed by a newline)
```

You are now ready to go – *ed* is waiting for you to tell it what to do. *Em* is started with the command

```
em
```

To edit an existing file, give

```
em filename
```

Creating Text – the Append command "a"

As our first problem, suppose we want to create some text starting from scratch. Perhaps we are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. *Em* also accepts their uppercase equivalents. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.) *Em* does have a prompt ">".

The first command is *append,* written as the letter

```
a
```

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, we just type an "a" followed by a newline, followed by the lines of text we want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that we have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "a" and "." aren't there, because they are not text.

To add more text to what we already have, just issue another "a" command, and continue typing.

Open mode

When using *em*, you probably will not want to use the append command. Instead, the *open mode* allows you to enter lines and to move around in the current line. To enter the open mode on an empty line, enter

o;

Now enter the lines of text you want, separated by returns as if you were appending text. If you make a typing error, you can backup the cursor by giving ^U, correct the error, and again advance the cursor by giving ^Q. (The combination ^Q means: press the control key, hit "q", and release the control key.) To stop entering text use ^D or ESCAPE; *em* will return with the usual prompt. There are other control characters to move around in a line, see the description in section I of the UNIX Programmers manual. Most important are the following:

group:	function:	11/60,GMX:	PDP 11/70
char:	advance	^Q	^C
	backup	^U	not available?
	delete	DELETE	DELETE
word:	advance	^A	^W
	backup	^B	^B
	delete	^W	^Z
line:	display to end	^E	^P
	reset to start	^S	^S
	delete backward	^X	^U
	delete forward	^F	^F
	re-type	^R	^A
exit:	with new text	^D	^D
	unchanged text	^C	^X
help:		^Y	^H

Other characters [including RETURN] are inserted as typed.

The "o;" command actually opens a new line after the current line. To open the current line, use "o", to open a new line before the current line, use "o-". The "o" command is like "c" in *ed*, The "o-" command is like "i".

Error Messages – "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by typing

?

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file – the Write command "w"

It's likely that we'll want to save our text for later use. To write out the contents of the buffer onto a file, we use the *write* command

w

followed by the filename we want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named "junk", for example, type

w junk

Leave a space between "w" and the file name. *Ed* will respond by printing the number of characters it wrote out. In our case, *ed* would respond with

    68

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing a file just makes a copy of the text – the buffer's contents are not disturbed, so we can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a "w" command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed – the Quit command "q"

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the "w" command, and then type the command

    q

which stands for *quit*. The system will respond with "%". At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. *Em* accepts the command "ok", meaning write and quit.

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using "w". Then leave *ed* with the "q" command, and print the file, to see that everything worked. (To print a file, say

    pr filename

or

    cat filename

in response to "%". Try both.)

Reading text from a file – the Edit command "e"

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the "w" command in a previous session. The *edit* command "e" fetches the entire contents of a file into the buffer. So if we had saved the three lines "Now is the time", etc., with a "w" command in an earlier session, the *ed* command

    e junk

would fetch the entire contents of the file "junk" into the buffer, and respond

    68

which is the number of characters in "junk". *If anything was already in the buffer, it is deleted first.*

If we use the "e" command to read a file into the buffer, then we need not use a file name after a subsequent "w" command; *ed* remembers the last file name used in an "e" command, and "w" will write on this file. Thus a common way to operate is

```
ed
e file
[editing session]
w
q
```

You can find out at any time what file name *ed* is remembering by typing the *file* command "f". In our case, if we typed

```
f
```

ed would reply

```
junk
```

Reading text from a file – the Read command "r"

Sometimes we want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command "r". The command

```
r junk
```

will read the file "junk" into the buffer; it adds it to the end of whatever is already in the buffer. So if we do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the "w" and "e" commands, "r" prints the number of characters read in, after the reading operation is complete.

Generally speaking, "r" is much less used than "e".

Exercise 2:

Experiment with the "e" command – try reading and printing various files. You may get an error "?", typically because you spelled the file name wrong. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer – the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, we use the print command

p

The way this is done is as follows. We specify the lines where we want printing to begin and where we want it to end, separated by a comma, and followed by the letter "p". Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) we say

1,2p (starting line=1, ending line=2 p)

Ed will respond with

Now is the time
for all good men

Suppose we want to print *all* the lines in the buffer. We could use "1,3p" as above if we knew there were exactly 3 lines in the buffer. But in general, we don't know how many there are, so what do we use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" – the dollar sign "$". Use it this way:

1,$p

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, give an interrupt; on most systems this is done with ^C. *Ed* will type

?

and wait for the next command.

To print the *last* line of the buffer, we could use

$,$p

but *ed* lets us abbreviate this to

$p

We can print any single line by typing the line number followed by a "p". Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

In fact, *ed* lets us abbreviate even further: we can print any single line by typing *just* the line number – no need to type the letter "p". So if we say

$

ed will print the last line of the buffer for us.

We can also use "$" in combinations like

$–1,$p

which prints the last two lines of the buffer. This helps when we want to see how far we got in typing.

Em has some convenient abvebrations for print commands:

print preceding twenty lines:	&
print next twenty lines:	"
print surrounding twenty lines:	%

so to step through your text a screen at a time, just type a number of double quotes "''".

Exercise 3:

As before, create some text using the append command and experiment with the "p" command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

```
3,1p
```

don't work.

The current line – "Dot" or "."

Suppose our buffer still contains the six lines as above, that we have just typed

```
1,3p
```

and *ed* has printed the three lines for us. Try typing just

```
p     (no line numbers).
```

This will print

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact it is the last (most recent) line that we have done anything with. (We just printed it!) We can repeat this "p" command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that we did anything to (in this case, line 3, which we just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.     (pronounced "dot").
```

Dot is a line number in the same way that "$" is; it means exactly "the current line", or loosely, "the line we most recently did something to." We can use it in several ways – one possibility is to say

```
.,$p
```

This will print all the lines from (including) the current line to the end of the buffer. In our case these are lines 3 through 6.

Some commands change the value of dot, while others do not. The print command sets dot to the number of the last line printed; by our last command, we would have "." = "$" = 6.

Dot is most useful when used in combinations like this one:

```
.+1  (or equivalently, .+1p)
```

This means "print the next line" and gives us a handy way to step slowly through a buffer. We can also say

```
.–1  (or .–1p )
```

which means "print the line *before* the current line." This enables us to go backwards if we wish. Another useful one is something like

.−3,.−1p

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

.=

Ed will respond by printing the value of dot.

Let's summarize some things about the "p" command and dot. Essentially "p" can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter "p"), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single newline will cause printing of the next line − it's equivalent to ".+1p". Try it. Try typing "^" − it's equivalent to ".−1p".

Deleting lines: the "d" command

Suppose we want to get rid of the three extra lines in the buffer. This is done by the *delete* command

d

Except that "d" deletes lines instead of printing them, its action is similar to that of "p". The lines to be deleted are specified for "d" exactly as they are for "p":

starting line, ending line d

Thus the command

4,$d

deletes lines 4 through the end. There are now three lines left, as we can check by using

1,$p

And notice that "$" now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to "$".

Exercise 4:

Experiment with "a", "e", "r", "w", "p", and "d" until you are sure that you know what they do, and until you understand how dot, "$", and line numbers are used.

If you are adventurous, try using line numbers with "a", "r", and "w" as well. You will find that "a" will append lines *after* the line number that you specify (rather than after dot); that "r" reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that "w" will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

0r filename

and you can enter lines at the beginning of the buffer by saying

0a
. . . *text* . . .

.

Notice that ".w" is *very* different from

.
w

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands – the substitute command

s

This is the command that is used to change individual words or letters within a line or group of lines. It is what we use, for example, for correcting spelling mistakes and typing errors. In *em* simple typing errors are best corrected using the open mode; the substitute command is only needed for global changes in the text.

Suppose that by a typing error, line 1 says

Now is th time

– the "e" has been left off "the". We can use "s" to fix this up as follows:

1s/th/the/

This says: "in line 1, substitute for the characters 'th' the characters 'the'." To verify that it works (*ed* will not print the result automatically) we say

p

and get

Now is the time

which is what we wanted. Notice that dot must have been set to the line where the substitution took place, since the "p" command printed that line. Dot is always set this way with the "s" command.

The general way to use the substitute command is

starting line, ending line s/*change this*/*to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between starting line and ending line. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for "p", except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error "?" as a warning.)

Thus we can say

1,$s/speling/spelling/

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the "s" command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, we can try again. (Notice that we put a print command on the same line as the substitute. With few exceptions, "p" can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/ . . . //

which means "change the first string of characters to *nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if we had

Nowxx is the time

we can say

s/xx//p

to get

Now is the time

Notice that "//" here means "no characters", not a blank. There *is* a difference! (See below for another meaning of "//".)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a
the other side of the coin
.
s/the/on the/p
```

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a "g" (for "global") to the "s" command, like this:

s/ . . . / . . . /gp

Try other characters instead of slashes to delimit the two sets of characters in the "s" command – anything should work except blanks or tabs.

(If you get funny results using any of the characters

^ . $ [* \

read the section on "Special Characters".)

Context searching – "/ . . . /"

With the substitute command mastered, we can move on to another highly important idea of *ed* – context searching.

Suppose we have our original three line text in the buffer:

Now is the time
for all good men
to come to the aid of their party.

Suppose we want to find the line that contains "their" so we can change it to "the". Now with only three lines in the buffer, it's pretty easy to keep track of what line the word "their" is on. But if the buffer contained several hundred lines, and we'd been making changes, deleting and rearranging lines, and so on, we would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way we say "search for a line that contains this particular string of characters" is to type

/string of characters we want to find/

For example, the *ed* line

/their/

is a context search which is sufficient to find the desired line – it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

"Next occurrence" means that *ed* starts looking for the string at line ".+1", searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from "$" to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

We can do both the search for the desired line *and* a substitution all at once, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression "/their/" is a context search expression. In their simplest form, all context search expressions are like this – a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like "s". We used them both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

/Now/+1
/good/
/party/−1

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, we could say

/Now/+1s/good/bad/

or

/good/s/good/bad/

or

/party/−1s/good/bad/

The choice is dictated only by convenience. We could print all three lines by, for instance

/Now/,/party/p

or

/Now/,/Now/+2p

or by any number of similar combinations. The first one of these might be better if we don't know how many lines are involved. (Of course, if there were only three lines in the buffer, we'd use

1,$p

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with "r", "w", and "a".)

Try context searching using "?text?" instead of "/text/". This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

^ . $ [* \

read the section on "Special Characters".)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

/string/

will find the next occurrence of "string". It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

//

This shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which will find the next occurrence of "string1" and replace it by "string2". This can save a lot of typing. Similarly

??

means "scan backwards for the same expression."

Em allows you to use "/" and "?" instead of "//" and "??".

Change and Insert – "c" and "i"

This section discusses the *change* command

c

which is used to change or replace a group of one or more lines, and the *insert* command

i

which is used for inserting a group of one or more lines.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines ".+1" through "$" to something else, type

.+1,$c
. . . *type the lines of text you want here* . . .
.

The lines you type between the "c" command and the "." will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the "c" command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of "." to end the input – this works just like the "." in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append – for instance

/string/i
. . . *type the lines to be inserted here* . . .
.

will insert the given text *before* the next line that contains "string". The text between "i" and "." is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

start, end d
i
. . . *text* . . .
.

is almost the same as

start, end c
. . . *text* . . .
.

These are not *precisely* the same if line "$" gets deleted. Check this out. What is dot?

Experiment with "a" and "i", to see that they are similar, but not the same. You will observe that

line number a
. . . *text* . . .
.

appends *after* the given line, while

line number i
. . . *text* . . .
.

inserts *before* it. Observe that if no line number is given, "i" inserts before line dot, while "a" appends after line dot.

Moving text around: the "m" command

The move command "m" is used for cutting and pasting – it lets you move a group of lines from one place to another in the buffer. Suppose we want to put the first three lines of the buffer at the end instead. We could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but we can do it a lot easier with the "m" command:

```
1,3m$
```

The general case is

start line, end line m *after this line*

Notice that there is a third line to be specified – the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if we had

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

we could reverse the two paragraphs like this:

```
/Second/,/second/m/First/-1
```

Notice the "−1" – the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command "g" is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain "peling". More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the "g" command does not give a "?" if "peling" is not found where the "s" command will.

There may be several commands (including "a", "c" "i" "r", "w", but not "g"); in that case, every line except the last must end with a backslash "\":

```
g/xxx/.-1s/abc/def/\
.+2s/ghi/jkl/\
.-2,.p
```

makes changes in the lines before and after each line that contains "xxx", then prints all three lines.

The "v" command is the same as "g", except that the commands are executed on every line that does *not* match the string following "v":

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ".", "*", "$", and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,*

/**x**.y/

means "a line with an x, *any character*, and a y," *not* just "a line with an x, a period, and a y." A complete list of the special characters that can cause trouble is the following:

^ . $ [* \ +

Note: The pattern matching mechanism of *ed* can do more than is explained here. There are some subtle differences between the pattern matching mechanisms of *ed* and *em*. Read the manual pages for a complete description. The character "+" is special in *em* only.
Warning: The backslash character \ is special to *ed*. For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change "\.*" into "backslash dot star".

Here is a hurried synopsis of the other special characters. First, the circumflex " ^ " signifies the beginning of a line. Thus

```
/^string/
```

finds "string" only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign "$" is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of "string" that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just "string", and

```
/^.$/
```

finds a line containing exactly one character.

The character ".", as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```
x+y
x y
x y
x.y
```

This is useful in conjunction with "*", which is a repetition character; "a*" is a shorthand for "any number of a's," so ".*" matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since ".*" finds the longest possible match, this goes up to the last comma.)

"[" is used with "]" to form "character classes"; for example,

```
/[1234567890]/
```

matches any single digit – any one of the characters inside the braces will cause a match.

Finally, the "&" is another shorthand character - it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

```
Now is the time
```

and we wanted to put parentheses around it. We could just retype the line, but this is tedious. Or we could say

```
s/^/(/
s/$/)/
```

using our knowledge of "^" and "$". But the easiest way uses the "&":

```
s/.*/(&)/
```

This says "match the whole line, and replace it by itself surrounded by parens." The "&" can be used several times in a line; consider using

```
s/.*/&?  &!!/
```

to produce

```
Now is the time?  Now is the time!!
```

We don't have to match the whole line, of course: if the buffer contains

```
the end of the world
```

we could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string "/world/" found the desired line; the shorthand "//" found the same word in the line; and the "&" saved us from typing it again.

The "&" is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. We can turn off the special meaning of "&" by preceding it with a "\":

```
s/ampersand/\&/
```

will convert the word "ampersand" into the literal symbol "&" in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of *e*, *r* and *w*, followed by a file name. Only one command is allowed per line, but a *p* command may follow any other command (except for *e*, *r*, *w* and *q*).

a (append) Add lines to the buffer (at line dot, unless a different line is specified). Appending continues until "." is typed on a new line. Dot is set to the last line appended.

c (change) Change the specified lines to the new text which follows. The new lines are terminated by a ".". If no lines are specified, replace line dot. Dot is set to last line changed.

d (delete) Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless "$" is deleted, in which case dot is set to "$".

e (edit) Edit new file. Any previous contents of the buffer are thrown away, so issue a *w* beforehand if you want to save them.

f (file) Print remembered filename. If a name follows *f* the remembered name will be set to it.

g (global) g/---/commands will execute the commands on those lines that contain "---", which can be any context search expression.

i (insert) Insert lines before specified line (or dot) until a "." is typed on a new line. Dot is set to last line inserted.

m (move) Move lines specified to after the line named after *m*. Dot is set to the last line moved.

o (open) Open a line for editing. *Em* only.

p (print) Print specified lines. If none specified, print line dot. A single line number is equivalent to "line-number p". A single newline prints ".+1", the next line.

q (quit) Exit from ed. Wipes out all text in buffer!!

r (read) Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s (substitute) s/string1/string2/ will substitute the characters of 'string2' for 'string1' in specified lines. If no line is specified, make substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. *s* changes only the first occurrence of string1 on a line; to change all of them, type a "g" after the final slash.

v (exclude) v/---/commands executes "commands" on those lines that *do not* contain "---".

w (write) Write out buffer onto a file. Dot is not changed.

.= (dot value) Print value of dot. ("=" by itself prints the value of "$".)

! (temporary escape)
Execute this line as a UNIX command.

/-----/ Context search. Search for next line which contains this string of characters. Print it. Dot is set to line where string found. Search starts at ".+1", wraps around from "$" to 1, and continues to dot, if necessary.

?-----? Context search in reverse direction. Start search at ".−1", scan to 1, wrap around to "$".

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

January 2, 1985

---

†UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX programmer's manual. The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *builtin* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or *words* interpreted as a *command name* followed by *arguments*. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a ^D which sent an end-of-file to the mail program. (Here and throughout this document, the notation "↑*x*" is to be read "control-*x*" and represents the striking of the *x* key while the control key is held down.) The mail program then echoed the characters 'EOT' and transmitted our message. The characters '% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '% ' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ↑D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal - the erase

character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the backspace (↑H) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

```
tset - e
```

which tells the program *tset* to set the erase character, and its default setting for this character is a backspace.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '- ' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current *working directory*. The option - *s* is the size option, and

```
ls - s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

    date > #now

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail*, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% '.

1.5. Input from files; pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

    sort < data

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

    sort data

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

    sort

then the sort program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ↑D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

    ls - s

normally produces a list of the files in our directory with the size of each in

---

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a ↑H, as we demonstrated in section 1.1 how this could be set up.

blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The - *n* option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls - s | sort - n
```

specifies that the output of the *ls* command run with the option - *s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the - *r* reverse sort option and the *head* command in combination with the previous command doing

```
ls - s | sort - n - r | head - 5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory*, which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

    prog.c prog.o prog.errs prog.output

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

    prog.*

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

    echo prog.*

will echo the names

    prog.c prog.errs prog.o prog.output

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

    echo ? ?? ???

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

    prog.[co]

will match

    prog.c prog.o

in the example above. We can also place two characters around a '-' in this notation to denote a range. Thus

    chap.[1-5]

might match files

    chap.1 chap.2 chap.3 chap.4 chap.5

if they existed. This is shorthand for

chap.[12345]

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list)* contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

No match.

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another users' login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp*. If I give the command

cp thatfile ~

the shell will expand this command to

cp thatfile /usr/bill

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

echo *

will not echo the character '*'. It will either echo an sorted list of filenames in the current *working directory*, or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '- ' in an argument word to a command is to enclose it

with single quotation characters ''', i.e.

    echo '*'

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within ''' characters. It and the character ''' itself can be preceded by a single '\' to prevent their special meaning. Thus

    echo \'\!

prints

    '!

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

    echo \''*'

which prints

    '*

since the first '\' escaped the first ''' and the '*' was enclosed between ''' characters.

1.8. Terminating commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

    cat /etc/passwd

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing the DEL or RUBOUT key on your terminal.* Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ↑D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ↑D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

    mail bill < prepared.text

the mail command will terminate without our typing a ↑D. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

---

*Many users use *stty* (1) to change the interrupt character to ↑C.

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a ↑Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
↑Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1] + Stopped          mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ↑Z. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a ↑D which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The ↑Z should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in section 2.6.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending

them a QUIT signal, sent by typing a ↑\. This will usually provoke the shell to produce a message like:

Quit (Core dumped)

indicating that a file 'core' has been created containing information about the program 'a.out's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See section 2.6 for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

cat /etc/passwd

command will, you can use the command

more /etc/passwd

The *more* program pauses after each complete screenful and types '- - More- - ' at which point you can hit a space to get another screenful, a return to get another line, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

cat /etc/passwd | more

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ↑S key to stop the typeout. The typeout will resume when you hit ↑Q or any other key, but ↑Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ↑S and ↑Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ↑O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ↑O is a toggle, so flushing can be turned off by typing ↑O again while output is being flushed.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

chsh myname /bin/csh

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it takes effect at next login.** You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so

you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
     'set noglob ; eval `tset - s - m dialup:c100rv4pna - m plugboard:?hp2621nl *`';
ts; stty intr ↑C kill ↑U crt
set time=15 history=10
msgs - f
if (- e $mail) then
     echo "${prompt}mail"
     mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ↑D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set;* this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ↑C and the line kill character to ↑U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '- f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print

'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

2.2. Shell variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

```
% set
argv    ()
cwd     /usr/bill
home    /usr/bill
path    (. /usr/ucb /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
term    c100rv4pna
user    bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this

means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

    rehash

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

    set ignoreeof

and to unset it do

    unset ignoreeof

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

    > filename

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

    set noclobber

in your *.login* file. Then trying to do

    date > now

would cause a diagnostic if 'now' existed already. You could type

    date >! now

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.†

2.3. The shell's history list

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our

†The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

```
% cat bug.c
main()

{
     printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
     printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
     printf("hello\n");
w
32
q
% !c - o bug
cc bug.c - o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls - l !*
ls - l a.out bug
-rwxr-xr-x 1 bill      3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill      3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ↑spp↑ssp
num bug.c | ssp
   1 main()
   3 {
   4        printf("hello\n");
   5 }
% !! | lpr
num bug.c | ssp | lpr
%
```

terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '- o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls - l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '↑' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmers Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

mail bill

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '- s'. We can do

alias ls ls - s

or even

alias dir ls - s

creating a new command syntax 'dir' which does an 'ls - s'. If we say

dir ~bill

then the shell will translate this to

ls - s /mnt/bill

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

alias cd ´cd \!* ; ls ´

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in '´' characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

alias whois ´grep \!↑ /etc/passwd´

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

command >& file

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

command |& lpr

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.#

Finally, it is possible to use the form

command >> file

to place output at the end of an existing file.†

2.6. Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

sort < data
ls - s | sort - n | head - 5
mail harold

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

du > usage &

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without out waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

#A command form

command >&! file

exists, and is used when *noclobber* is set and *file* already exists.
†If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

command >>! file

makes it not be an error for file to not exist when *noclobber* is set.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] - Done              du > usage
%
```

If the job did not terminate normally the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the *notify* variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are re-used.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls - s | sort - n > usage &
[2] 2034 2035
%
```

runs the 'ls' program with the '- s' options, pipes this output into the 'sort' program with the '- n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediates prompts for a new command, leaving the job running simultaneously.

As mentioned in section 1.8, foreground jobs become *suspended* by typing ↑Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
↑Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)   sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the *bg* command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
↑Z
Stopped
% bg
[1] du > usage &
%
```

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job - identified by a '- ' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%- ' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '- l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls - s | sort - n > myfile &
[2] 3405
% mail bill
↑Z
Stopped
% jobs
[1] – Running          du > usage
[2]   Running          ls - s | sort - n > myfile
[3] + Stopped          mail bill
% fg %ls
ls - s | sort - n > myfile
% more myfile
```

The *fg* command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

```
% kill %1
[1] Terminated          du > usage
%
```

The *notify* command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```
% ed bigfile
120000
1,$s/thisword/thatword/
↑Z
Stopped
% bg
[1] ed bigfile &
%
 . . . some foreground commands
[1] Stopped (tty input) ed bigfile
% fg
ed bigfile
w
```

```
120000
q
%
```

So after the 's' command was issued, the 'ed' job was stopped with ↑Z and then put in the background using *bg*. Some time later when the 's' command was finished, *ed* tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The *fg* command returned the 'ed' job to the foreground where it could once again accept commands from the terminal.

The command

```
stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)wc hugefile
% fg wc
wc hugefile
  13371  30123  302577
% stty - tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the *jobs* command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The *ps* can be used in this case to find out about background jobs not started in the current shell.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir*, creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory *newpaper*. where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just

```
cd
```

with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable *cwd*. The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command *pushd* is used in place of the *cd* command, the shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this list at any time by typing the 'directories' command *dirs*.

```
% pushd newpaper/references
~/newpaper/references  ~
% pushd /usr/lib/tmac
/usr/lib/tmac  ~/newpaper/references  ~
% dirs
/usr/lib/tmac  ~/newpaper/references  ~
% popd
~/newpaper/references  ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory–in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than

pwd since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *csh* manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs - l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
↑Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no cd command was issued. In the above example the 'ed' job was still in '/mnt/bill/project' even though the shell had changed to '/mnt/bill'. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd: ~/myproject)
 . . . after some editing
q
(wd now: ~)
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The '- l' option of *jobs* will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

2.8. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

alias ls

to show the current alias for, e.g., 'ls'.

The *echo* command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*. By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

set prompt='\! % '

Note that the '!' character had to be *escaped* here even within '' characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

```
cputime       unlimited
filesize      unlimited
datasize      5616 kbytes
stacksize     512 kbytes
coredumpsizeunlimited
```

Limits can be set, e.g.:

limit coredumpsize 128k

Most reasonable units abbreviations will work; see the *csh* manual page for more details.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

repeat 5 cat one >> five

The *setenv* command can be used to set variables in the environment. Thus

setenv TERM adm3a

will set the value of the environment variable TERM to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
   52   178   1347 /etc/rc
   52   178   1347 /usr/bill/rc
  104   356   2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be

unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
$argv[$#argv]
```

would give 'c', and

```
$argv[1- 2]
```

would give 'a b'. Other notations useful in shell scripts are

$*n*

where *n* is an integer as a shorthand for

$argv[*n*]

the *n th* parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (↑D) was typed.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if *n* is not in the range '1- $#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n- '; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm- n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '==' and '!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
- ? filename
```

where '?' is replace by a number of single characters. For instance the expression primitive

```
- e filename
```

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

      if ($i !~ *.c) continue  # not a .c file so do nothing

      if (! - r ~/backup/$i:t) then
            echo $i:t not in backup... not cp\'ed
            continue
      endif

      cmp - s $i ~/backup/$i:t # to set $status

      if ($status != 0) then
            echo new backup of $i
            cp $i ~/backup/$i:t
      endif
end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

if (expression) **then**
    command
    ...
endif

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

---

†The following two formats are not currently acceptable to the shell:

if (expression)   # **Won't work!**
then
        command
        ...

The shell does have another form of the if statement of the form

if (expression) **command**

which can be written

if (expression) \
    command

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif*, e.g.:

if (expression) **then**
    commands
else if (expression) **then**
    commands
...

else
    commands
endif

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier ':r' here to extract a root of a filename or ':e' to extract the *extension*. Thus if the variable *i* has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ':r' modifier strips off the trailing '.bar' and the the ':e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the programmers manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.# Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell.

---

endif

and

if (expression) **then** command **endif**   # **Won't work**

#It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

This character can be quoted using '"' or '\' to place it in an argument word.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

> **while** (expression)
>   commands
> **end**

and

> **switch** (word)
>
> **case** str1:
>   commands
>   **breaksw**
>
> ...
>
> **case** strn:
>   commands
>   **breaksw**
>
> **default:**
>   commands
>   **breaksw**
>
> **endsw**

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

> loop:
>   commands
>   **goto** loop

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank - - remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in '' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *- v* and *- x* command line options can be used to help trace the actions of the shell. The *- n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '' does.

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep - c csh$ /etc/passwd
27
% grep - c nsh$ /etc/passwd
128
% grep - c - v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '- v sh$')
? grep - c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

    A{str1,str2,...strn}B

expands to

    Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

    mkdir ~/{hdrs,retrofit,csh}

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

    chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how−ex}}

4.3. Command substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

    set pwd=`pwd`

to save the current directory in the variable *pwd* or to do

    ex `grep - l TRACE *.c`

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

---

*Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

Appendix - Special characters

The following table lists the special characters of *csh* and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic metacharacters

- ; 2.4 separates commands to be executed sequentially
- | 1.5 separates commands in a pipeline
- ( ) 2.2,3.6 brackets expressions and variable values
- & 2.5 follows commands to be executed without waiting for completion

Filename metacharacters

- / 1.6 separates components of a file's pathname
- ? 1.6 expansion character matching any single character
- * 1.6 expansion character matching any sequence of characters
- [ ] 1.6 expansion sequence matching any single character from a set
- ~ 1.6 used at the beginning of a filename to indicate home directories
- { } 4.2 used to specify groups of arguments with common parts

Quotation metacharacters

- \ 1.7 prevents meta-meaning of following single character
- ' 1.7 prevents meta-meaning of a group of characters
- " 4.3 like ', but allows variable and command expansion

Input/output metacharacters

- < 1.5 indicates redirected input
- > 1.3 indicates redirected output

Expansion/substitution metacharacters

- $ 3.4 indicates variable substitution
- ! 2.3 indicates history substitution
- : 3.6 precedes substitution modifiers
- ↑ 2.3 used in special forms of history substitution
- ` 4.3 indicates command substitution

Other metacharacters

- # 1.3,3.6 begins scratch file names; indicates shell comments
- - 1.2 prefixes option (flag) arguments to commands
- % 2.6 prefixes job name specifications

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX programmer's manual in section 1. You can get an online copy of its manual page by doing

man 1 pr

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

. Your current directory has the name '.' as well as the name printed by the command *pwd*; see also *dirs*. The current directory '.' is usually the first *component* of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating *components* of filenames (1.6). The character '.' at the beginning of a *component* of a *pathname* is treated specially and not matched by the *filename expansion* metacharacters '?', '*', and '[' ']' pairs (1.6).

.. Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir*, i.e.

chdir paper

you can return to the parent directory by doing

chdir ..

The current directory is printed by *pwd* (2.7).

a.out Compilers which create executable images create them, by default, in the file *a.out*. for historical reasons (2.3).

absolute pathname
A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system - called the *root* directory. *Pathname*s which are not *absolute* are called *relative* (see definition of *relative pathname*) (1.6).

alias An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (2.4).

argument Commands in UNIX receive a list of *argument* words. Thus the command

echo a b c

consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (1.1).

argv The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).

background — Commands started without waiting for them to complete are called *background* commands (2.6).

base — A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* - the part after the '.'. See *filename* and *extension* (1.6)

bg — The *bg* command causes a *suspended* job to continue execution in the *background* (2.6).

bin — A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

break — *Break* is a builtin command used to exit from loops within the control structure of the shell (3.7).

breaksw — The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (3.7).

builtin — A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

case — A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh(1)' (3.7).

cat — The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).

cd — The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory (2.4, 2.7).

chdir — The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

chsh — The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing

```
chsh your-login-name /bin/csh
```

Thus I would do

```
chsh bill /bin/csh
```

It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).

cmp — *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.

command — A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory within the UNIX system, is called a *command* (1.1).

command name
: When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution
: The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (4.3).

component — A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable which has multiple strings as value is said to have several *component* s; each string is a *component* of the variable.

continue — A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

control- — Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control*-c is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an up-arrow (↑) followed by the corresponding letter when you type a *control* character (e.g. '↑C' for *control*-c (1.8).

core dump — When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form

    Illegal instruction (core dumped)

(where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

cp — The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).

csh — The name of the shell program that this document describes.

.cshrc — The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).

cwd — The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).

date — The *date* command prints the current date and time (1.3).

debugging — *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).

default: — The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).

DELETE — The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ↑C.

detached — A command that continues running in the *background* after you logout is said to be *detached*.

diagnostic — An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output*, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).

directory — A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).

directory stack — The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left (2.7).

dirs — The *dirs* command prints the shell's *directory stack* (2.7).

du — The *du* command is a program (described in 'du(1)') which prints the number of disk blocks is all directories below and including your current *working directory* (2.6).

echo — The *echo* command prints its arguments (1.6, 3.6).

else — The *else* command is part of the 'if-then-else-endif' control command construct (3.6).

endif — If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true (3.6).

EOF — An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an *end-of-file* when the command sending them input completes. Most commands terminate when they receive an *end-of-file*. The shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing

too many control-d's (1.1, 1.8, 3.8).

escape
: A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

    echo *

will echo the character '*' while just

    echo *

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended*. Most systems use control-s to stop the output and control-q to start it.

/etc/passwd
: This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters (1.8). You can look at this file by saying

    cat /etc/passwd

The commands *finger* and *grep* are often used to search for information in this file. See 'finger(1)', 'passwd(5)', and 'grep(1)' for more details.

exit
: The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).

exit status
: A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script to give a non-zero *exit status* (3.6).

expansion
: The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion*. Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions* (1.6, 3.4, 4.2).

expressions
: *Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (3.5).

extension
: Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '-me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).

fg — The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (1.8, 2.6).

filename — Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension* (1.6).

filename expansion — *Filename expansion* uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (1.6, 4.2).

flag — Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '- ' (1.2). Thus the *ls* (list files) command has an option '- s' to list the sizes of files. This is specified

```
ls - s
```

foreach — The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

foreground — When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground*. This is as opposed to *background*. *Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard (1.8, 2.6).

goto — The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

grep — The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file */etc/passwd* which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed(1)' and 'ex(1)'. *Grep* stands for 'globally find *regular expression* and print' (2.4).

head — The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).
Head is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).

history — The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (2.3).

home directory — Each user has a *home directory*, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~' (1.6).

if — A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).

ignoreeof — Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).

input — Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in *pipelines* will read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).

interrupt — An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key (although users can and often do change the interrupt character, usually to ↑C). It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (1.8, 3.9).

job — One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended* (2.6).

job control — The builtin functions that control the execution of jobs are called *job control* commands. These are *bg*, *fg*, *stop*, *kill* (2.6).

job number — When each job is started it is assigned a small number called a *job number* which is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job (2.6).

jobs — The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (2.6).

kill — A command which sends a signal to a job causing it to terminate (2.6).

.login — The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (2.1).

login shell — The shell that is started on your terminal when you login is called your *login shell*. It is different from other shells which you may run (e.g. on shell scripts) in that it reads the *.login* file before reading commands from the terminal and it reads the *.logout* file after you logout (2.1).

logout — The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file*, but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.8).

.logout — When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.

lpr — The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).

ls — The *ls* (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail — The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.1).

make — The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).

makefile — The file containing commands for *make* is called *makefile* (3.2).

manual — The *manual* often referred to is the 'UNIX programmer's manual'. It contains a number of sections and a description of each UNIX program. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

metacharacter
: Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a *metacharacter* is the character '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words (1.4). The appendix to this user's manual lists the *metacharacters* in groups by their function.

mkdir
: The *mkdir* command is used to create a new directory.

modifier
: Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

more
: The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

noclobber
: The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).

noglob
: The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (3.6).

notify
: The *notify* command tells the shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the shell to always report the termination of *background* jobs exactly when they occur (2.6).

onintr
: The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (3.9).

output
: Many commands in UNIX result in some lines of text which are called their *output*. This *output* is usually placed on what is known as the *standard output* which is normally connected to the user's terminal. The shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|' it is also possible for the *standard output* of one command to become the *standard input* of another command (1.5). Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the *standard output* has been sent to a file or another command.

but it is possible to direct error diagnostics along with *standard output* using a special metanotation (2.5).

pushd — The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).

path — The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

    path(. /usr/ucb /bin /usr/bin)

the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

    chmod 755 script

was executed to turn this execute permission on (3.3). If you add new commands to a directory in the *path*, you should issue the command *rehash* (2.2).

pathname — A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* may name a directory, but usually names a file.

pipeline — A group of commands which are connected together, the *standard output* of each connected to the *standard input* of the next, is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3).

popd — The *popd* command changes the shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so (2.7).

port — The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr — The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

printenv — The *printenv* command is used to print the current setting of variables in the environment (2.8).

process — An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started - called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.

program — Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.

programmer's manual
: Also referred to as the *manual*. See the glossary entry for 'manual'.

prompt — Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex(1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).

ps — The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

pwd — The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.

quit — The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).

quotation — The process by which metacharacters are prevented their special meaning, usually by using the character ' in pairs, or by using the character '\', is referred to as *quotation* (1.7).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

rehash — The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).

relative pathname
: A *pathname* which does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to

some file or directory in the *working directory*, and subsequent *components* between '/' characters refer to directories below the *working directory*. *Pathnames* that are not *relative* are called *absolute pathnames* (1.6).

repeat — The *repeat* command iterates another command a specified number of times.

root — The directory that is at the top of the entire directory structure is called the *root* directory since it is the 'root' of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is '/'. *Pathnames* starting with '/' are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation (1.6).

RUBOUT — The RUBOUT or DELETE key sends an interrupt to the current job. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell. Users often change interrupt to be generated by ↑C rather than DELETE by using the *stty* command.

scratch file — Files whose names begin with a '#' are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).

script — Sequences of shell commands placed in a file are called shell command *scripts*. It is often possible to perform simple tasks using these *scripts* without writing a program in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).

set — The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

setenv — Variables in the environment 'environ(5)' can be changed by using the *setenv* builtin command (2.8). The *printenv* command can be used to print the value of the variables in the environment.

shell — A *shell* is a command language interpreter. It is possible to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular *shell*, called *csh*.

shell script — See *script* (3.3, 3.10).

signal — A *signal* in UNIX is a short message that is sent to a running program which causes something to happen to that process. *Signals* are sent either by typing special *control* characters on the keyboard or by using the *kill* or *stop* commands (1.8, 2.6).

sort — The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (1.5).

source — The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.8).

special character
: See *metacharacters* and the appendix to this manual.

standard
: We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).

status
: A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (3.6).

stop
: The *stop* command causes a *background* job to become *suspended* (2.6).

string
: A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters (2.2).

stty
: The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty(1)' for a complete description (2.6).

substitution
: The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (3.4).

suspended
: A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control*-z at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command (2.6).

switch
: The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).

termination
: When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end-of-file* from their *standard input*. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified jobs (2.6).

then
: The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time
: The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command (2.1, 2.8).

tset
: The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).

tty
: The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.

unalias — The *unalias* command removes aliases (2.8).

UNIX — UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.

unset — The *unset* command removes the definitions of shell variables (2.2, 2.8).

variable expansion — See *variables* and *expansion* (2.2, 3.4).

variables — *Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing shell programs (shell command scripts) (2.2).

verbose — The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's - *v* command line option (3.10).

wc — The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).

while — The *while* builtin control construct is used in shell command scripts (3.7).

word — A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '- ', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with '´' characters except for the characters '´' and '!' which require special treatment (1.1). This process of placing special characters in *words* without their special meaning is called *quoting*.

working directory — At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.

write — The *write* command is used to communicate with other users who are logged in to UNIX.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed.*

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Sed is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

- **-n:** tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- **-e:** tells *sed* to take the next argument as an editing command;
- **-f:** tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }') (Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '*d*' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address,

and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/ matches lines 1, 3, 4 in our sample text
/an.*an/ matches line 1
/^an/ matches no lines
/./ matches all lines
/\./ matches line 5
/r*an/ matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/ matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n       n
i\      c\
XXXX    XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

> & is replaced by the string matched by <pattern>
>
> *d* (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\\(' and '\\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\\(').
>
> As in patterns, special characters may be made literal by preceding them with backslash ('\\').

The <flags> argument may contain the following flags:

> g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.
>
> p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.
>
> w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.
>
> A single space must separate *w* and <filename>.
>
> The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.
>
> A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

s/to/by/w changes

produces, on the standard output:

In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.

and, on the file 'changes':

Through caverns measureless by man
Down by a sunless sea.

If the nocopy option is in effect, the command:

s/[.,;?:]/*P&*/gp

produces:

A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*

Finally, to illustrate the effect of the *g* flag, the command:

/X/s/an/AN/p

produces (assuming nocopy mode):

In XANadu did Kubhla Khan

and the command:

/X/s/an/AN/gp

produces:

In XANadu did Kubhla KhAN

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a*

functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

> Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
        Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
        and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
        Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

> The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

> Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

> Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

DOCUMENTS FOR USE WITH THE
UNIX TIME-SHARING SYSTEM

Hybrid PWB - V7

The enclosed UNIX documentation is supplied
in accordance with the Software Agreement
you have with the Western Electric Company.

Western Electric

Guilford Center
P. O. Box 25000
Greensboro, NC 27420
919 697-2000

Patent Licensing

OCT 03 1980

TECHNISCHE HOGESCHOOL DELFT
Department of Mathematics
Juianalaan 132
2628 BL Delft
The Netherlands

Attn: Mr. P. J. van der Hoff

Gentlemen:

Re: May 1, 1979 Software Agreement Between Us
Relating to PWB/UNIX* Time Sharing Operating
System

In response to the August 22, 1980 request from Mr. van der Hoff, your institution may use the licensed software pursuant to the referenced agreement on the following additional CPU's:

LSI-11, Serial No. WM 1720
PDP 11/60, Serial No. AG 00373
Technische Hogeschool Delft
Department of Mathematics - Comp. Rm. 0.101
Julianalaan 132
2628 BL Delft
The Netherlands

Yours truly,

O. L. Wilson

O. L. WILSON
Patent Licensing Manager

*UNIX is a trademark of Bell Laboratories.

Copyright 1979, Bell Telephone Laboratories, Incorporated. Holders of a UNIX™ software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

CONTENTS

1. Typing Documents on the UNIX System
2. A Guide to Preparing Documents with -ms
3. NROFF/TROFF User's Manual
4. A TROFF Tutorial
5. PWB/MM Programmer's Workbench Memorandum Macros
6. Typing Documents with PWB/MM
7. TBL - A Program to Format Tables
8. A System for Typesetting Mathematics
9. Typesetting Mathematics - User's Guide (Second Edition)

Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction. This memorandum describes a package of commands to produce papers using the *troff* and *nroff* formatting programs on the UNIX system. As with other *roff*-derived programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. The commands available in this package are listed in Appendix A.

Text. Type normally, except that instead of indenting for paragraphs, place a line reading ".PP" before each paragraph. This will produce indenting and extra space.

Alternatively, the command .LP that was used here will produce a left-aligned (block) paragraph. The paragraph spacing can be changed: see below under "Registers."

Beginning. For a document with a paper-type cover sheet, the input should start as follows:

```
[optional overall format .RP — see below]
.TL
Title of document (one or more lines)
.AU
Author(s) (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE (abstract end)
text ... (begins with .PP, which see)
```

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ".AB no" for ".AB". Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. ***Warning:*** You can't just begin a document with a line of text. Some −ms command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TL, .SH, .NH is good enough. Figure 1 shows the legal arrangement of commands at the start of a document.

Cover Sheets and First Pages. The first line of a document signals the general format of the first page. In particular, if it is ".RP" a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

In general −ms is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored.

Warning: don't put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed in them may not behave as you expect. Don't forget that some −ms command must precede any input text.

Page headings. The −ms macros, by default, will print a page heading containing a page number (if greater than 1). A default page footer is provided only in *nroff*, where the date is used. The user can make minor adjustments to the page headings/footings by redefining the strings LH, CH, and RH which are the left, center and right portions of the page headings, respectively; and the strings LF, CF, and RF, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros PT and BT, which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. The user who redefines these macros should be careful not to change parameters such as point size or font without resetting them to default values.

Multi-column formats. If you place the command ".2C" in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command ".1C" will go back to one-column format and also skip to a new page. The ".2C" command is actually a special case of the command

    .MC [column width [gutter width]]

which makes multiple columns with the specified column and gutter width; as many columns as will fit across the page are used. Thus triple, quadruple, ... column pages can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns) a new page is started.

Headings. To produce a special heading, there are two commands. If you type

    .NH
    type section heading here
    may be several lines

you will get automatically numbered section headings (1, 2, 3, ...), in boldface. For example,

    .NH
    Care and Feeding of Department Heads

produces

1. Care and Feeding of Department Heads

Alternatively,

    .SH
    Care and Feeding of Directors

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with .PP or .LP, indicating the end of the heading. Headings may contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a "level" number and an appropriate subsection number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

    .NH
    Erie-Lackawanna
    .NH 2
    Morris and Essex Division
    .NH 3
    Gladstone Branch
    .NH 3
    Montclair Branch
    .NH 2
    Boonton Line

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

An explicit ".NH 0" will reset the numbering of level 1 to one, as here:

    .NH 0
    Penn Central

1. Penn Central

Indented paragraphs. (Paragraphs with hanging numbers, e.g. references.) The sequence

```
.IP [1]
Text for first paragraph, typed
normally for as long as you would
like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as you would like on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs may be followed by an ordinary paragraph beginning with .PP or .LP, depending on whether you wish indenting or not. The command .LP was used here.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations or
such matter.
.LP
```

will produce

> This material will just be turned into a block indent suitable for quotations or such matter.

If a non-standard amount of indenting is required, it may be specified after the label (in character positions) and will remain in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces this:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as "move right" and the .RE command as "move left". As an example

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

will result in

1. Bell Laboratories
   - 1.1 Murray Hill
   - 1.2 Holmdel
   - 1.3 Whippany
     - 1.3.1 Madison
   - 1.4 Chester

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

> A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) should be bracketed with .QS and .QE.

Emphasis. To get italics (on the typesetter) or underlining (on the terminal) say

```
.I
as much text as you want
can be typed here
.R
```

as was done for *these three words.* The .R command restores the normal (usually Roman) font. If only one word is to be italicized, it may be just given on the line with the .I command,

```
.I word
```

and in this case no .R is needed to restore the previous font. **Boldface** can be produced by

```
.B
Text to be set in boldface
goes here
.R
```

and also will be underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on the same line as the .B command.

A few size changes can be specified similarly with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for increased effect (here one .NL canceled two .SM commands).

If actual underlining as opposed to italicizing is required on the typesetter, the command

```
.UL word
```

will underline a word. There is no way to underline multiple words on the typesetter.

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

Displays and Tables. To prepare displays of lines, such as tables, in which the lines should not be re-arranged, enclose them in the commands .DS and .DE

```
.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE
```

By default, lines between .DS and .DE are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged. A plain .DS is equivalent to .DS I, which indents and left-adjusts. Thus,

these lines were preceded
by .DS C and followed by
a .DE command;

whereas

these lines were preceded
by .DS L and followed by
a .DE command.

Note that .DS C centers each line; there is a variant .DS B that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .DS command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

Boxing words or lines. To draw rectangular boxes around words the command

```
.BX word
```

will print word as shown. The boxes will not be neat on a terminal, and this should not be used as a substitute for italics.

Longer pieces of text may be boxed by enclosing them with .B1 and .B2:

```
.B1
text...
.B2
```

as has been done here.

Keeping blocks together. If you wish to keep a table or other block of lines together on a page, there are "keep -

* Like this.

release" commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it will begin on a new page. Lines bracketed by .DS and .DE commands are automatically kept together this way. There is also a "keep floating" command: if the block to be kept together is preceded by .KF instead of .KS and does not fit on the current page, it will be moved down through the text until the top of the next page. Thus, no large blank space will be introduced in the document.

Nroff/Troff commands. Among the useful commands from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

.bp - begin new page.
.br - "break", stop running text
from line to line.
.sp n - insert n blank lines.
.na - don't adjust right margins.

Date. By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don't. To force the date, say ".DA". To force no date, say ".ND". To lie about the date, say ".DA July 4, 1776" which puts the specified date at the bottom of each page. The command

.ND May 8, 1945

in ".RP" format places the specified date on the cover sheet and nowhere else. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors' names will be output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

Registers. Certain of the registers used by −ms can be altered to change default settings. They should be changed with .nr commands, as with

.nr PS 9

to make the default point size 9 point. If the effect is needed immediately, the normal *troff* command should be used in addition to changing the number register.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6"
LT	title length	next para.	6"
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27"
HM	top margin	next page	1"
FM	bottom margin	next page	1"

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

Accents. To simplify typing certain foreign words, strings representing common accent marks are defined. They precede the letter over which the mark is to appear. Here are the strings:

Input	Output	Input	Output
*'e	é	*~a	ã
*`e	è	*Ce	ě
*:u	ü	*,c	ç
*^e	ê		

Use. After your document is prepared and stored on a file, you can print it on a terminal with the command*

nroff −ms file

and you can print it on the typesetter with the command

troff −ms file

(many options are possible). In each case, if your document is stored in several files, just list all the filenames where we have used "file". If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

* If .2C was used, pipe the *nroff* output through *col;* make the first line of the input ".pi /usr/bin/col."

References and further study. If you have to do Greek or mathematics, see ***eqn*** [1] for equation setting. To aid ***eqn*** users, ***−ms*** provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

for a left-adjusted equation numbered (1.3a).

Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros.

To learn more about *troff* see [3] for a general introduction, and [4] for the full details (experts only). Information on related UNIX commands is in [5]. For jobs that do not seem well-adapted to −ms, consider other macro packages. It is often far easier to write a specific macro packages for such tasks as imitating particular journals than to try to adapt −ms.

Acknowledgment. Many thanks are due to Brian Kernighan for his help in the design and implementation of this package, and for his assistance in preparing this manual.

References

[1] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics — Users Guide (2nd edition),* Bell Laboratories Computing Science Report no. 17.

[2] M. E. Lesk, *Tbl — A Program to Format Tables,* Bell Laboratories Computing Science Report no. 45.

[3] B. W. Kernighan, *A Troff Tutorial,* Bell Laboratories, 1976.

[4] J. F. Ossanna, *Nroff/Troff Reference Manual,* Bell Laboratories Computing Science Report no. 51.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, 1978.

Appendix A
List of Commands

1C	Return to single column format.
2C	Start double column format.
AB	Begin abstract.
AE	End abstract.
AI	Specify author's institution.
AU	Specify author.
B	Begin boldface.
DA	Provide the date on each page.
DE	End display.
DS	Start display (also CD, LD, ID).
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
ND	Change or cancel date.
NH	Specify numbered heading.
NL	Return to normal type size.
PP	Begin paragraph.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TL	Specify title.
UL	Underline one word.

Register Names

The following register names are used by −ms internally. Independent use of these names in one's own macros may produce incorrect output. Note that no lower case letters are used in any −ms internal name.

Number registers used in −ms

:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in −ms

'	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
^	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
~	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

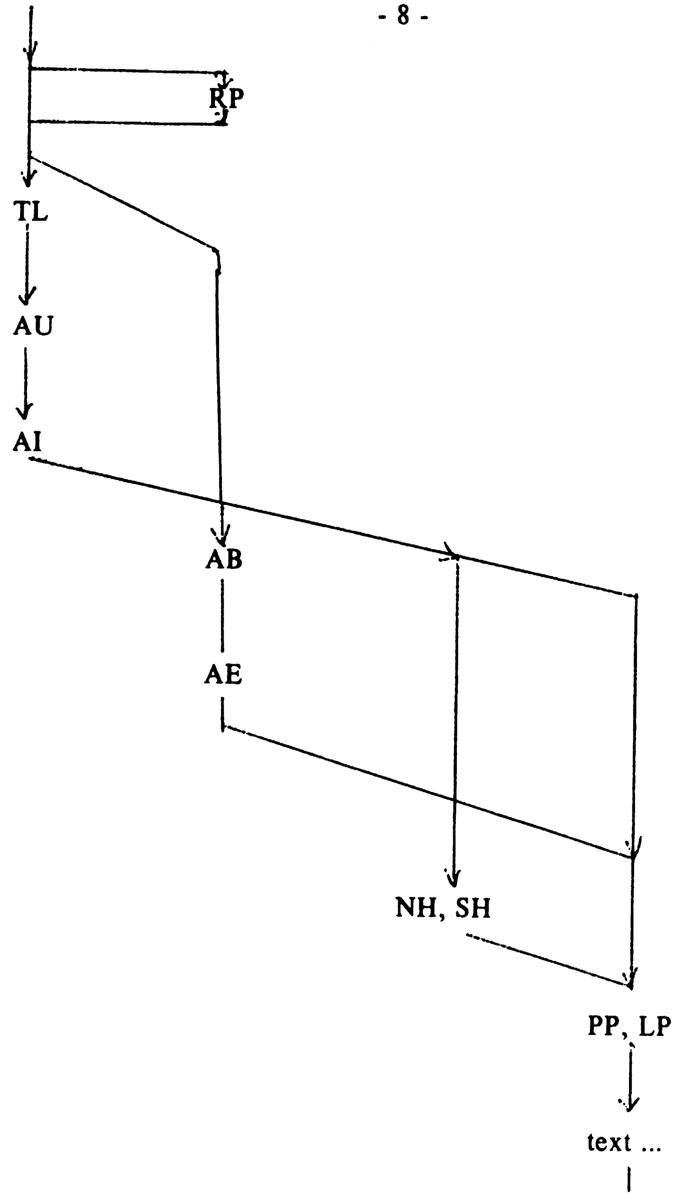


Figure 1

A Guide to Preparing Documents with −ms

M. E. Lesk

Bell Laboratories

August 1978

This guide gives some simple examples of document preparation on Bell Labs computers, emphasizing the use of the *−ms* macro package. It enormously abbreviates information in

1. *Typing Documents on UNIX and GCOS,* by M. E. Lesk;
2. *Typesetting Mathematics — User's Guide,* by B. W. Kernighan and L. L. Cherry; and
3. *Tbl — A Program to Format Tables,* by M. E. Lesk.

These memos are all included in the *UNIX Programmer's Manual, Volume 2.* The new user should also have *A Tutorial Introduction to the UNIX Text Editor,* by B. W. Kernighan.

For more detailed information, read *Advanced Editing on UNIX* and *A Troff Tutorial,* by B. W. Kernighan, and (for experts) *Nroff/Troff Reference Manual* by J. F. Ossanna. Information on related commands is found (for UNIX users) in *UNIX for Beginners* by B. W. Kernighan and the *UNIX Programmer's Manual* by K. Thompson and D. M. Ritchie.

Contents

A TM 2
A released paper 3
An internal memo, and headings . . . 4
Lists, displays, and footnotes 5
Indents, keeps, and double column . 6
Equations and registers 7
Tables and usage 8

Throughout the examples, input is shown in
  this Helvetica sans serif font
while the resulting output is shown in
  this Times Roman font.

UNIX Document no. 1111

Commands for a TM

```
.TM 1978-5b3 99999 99999-11
.ND April 1, 1976
.TL
The Role of the Allen Wrench in Modern
Electronics
.AU "MH 2G-111" 2345
J. Q. Pencilpusher
.AU "MH 1K-222" 5432
X. Y. Hardwired
.AI
.MH
.OK
Tools
Design
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It must attract the reader into sending for
the complete memorandum.
.AE
.CS 10 2 12 5 6 7
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.SG MH-1234-JQP/XYH-unix
.NH
References ...
```

Commands not needed in a particular format are ignored.

Bell Laboratories Cover Sheet for TM

This information is for employees of Bell Laboratories. (GEI 13.9-3)

Title- **The Role of the Allen Wrench in Modern Electronics** Date- **April 1, 1976**

TM- **1978-5b3**

Other Keywords- **Tools Design**

Author	Location	Ext.	
J. Q. Pencilpusher	MH 2G-111	2345	Charging Case- **99999**
X. Y. Hardwired	MH 1K-222	5432	Filing Case- **99999a**

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

Pages Text 10	Other 2	Total 12
No. Figures 5	No. Tables 6	No. Refs. 7

E-1932-U (6-73) SEE REVERSE SIDE FOR DISTRIBUTION LIST

A Released Paper with Mathematics

```
.EQ
delim $$
.EN
.RP

... (as for a TM)

.CS 10 2 12 5 6 7
.NH
Introduction
.PP
The solution to the torque handle equation
.EQ (1)
sum from 0 to inf F ( x sub i ) = G ( x )
.EN
is found with the transformation $ x = rho over
theta $ where $ rho = G prime (x) $ and $theta$
is derived ...
```

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

April 1, 1976

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

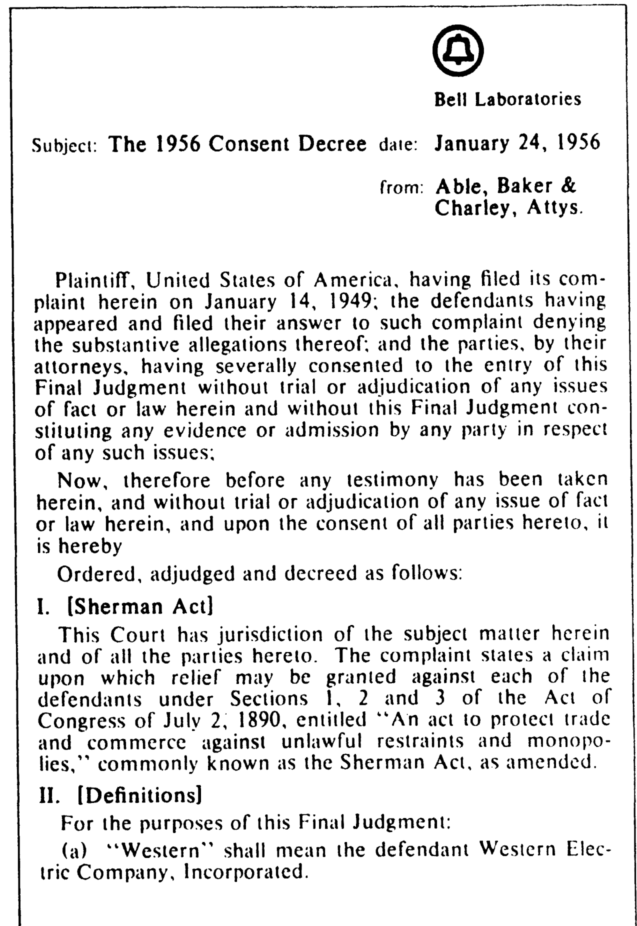
The solution to the torque handle equation

$$\sum_{0}^{\infty} F(x_i) = G(x) \quad (1)$$

is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived from well-known principles.

An Internal Memorandum

```
.IM
.ND January 24, 1956
.TL
The 1956 Consent Decree
.AU
Able, Baker &
Charley, Attys.
.PP
Plaintiff, United States of America, having filed
its complaint herein on January 14, 1949; the
defendants having appeared and filed their
answer to such complaint denying the
substantive allegations thereof; and the parties,
by their attorneys, ...
```



Bell Laboratories

Subject: **The 1956 Consent Decree** date: **January 24, 1956**

from: **Able, Baker & Charley, Attys.**

Plaintiff, United States of America, having filed its complaint herein on January 14, 1949; the defendants having appeared and filed their answer to such complaint denying the substantive allegations thereof; and the parties, by their attorneys, having severally consented to the entry of this Final Judgment without trial or adjudication of any issues of fact or law herein and without this Final Judgment constituting any evidence or admission by any party in respect of any such issues;

Now, therefore before any testimony has been taken herein, and without trial or adjudication of any issue of fact or law herein, and upon the consent of all parties hereto, it is hereby

Ordered, adjudged and decreed as follows:

I. [Sherman Act]

This Court has jurisdiction of the subject matter herein and of all the parties hereto. The complaint states a claim upon which relief may be granted against each of the defendants under Sections 1, 2 and 3 of the Act of Congress of July 2, 1890, entitled "An act to protect trade and commerce against unlawful restraints and monopolies," commonly known as the Sherman Act, as amended.

II. [Definitions]

For the purposes of this Final Judgment:

(a) "Western" shall mean the defendant Western Electric Company, Incorporated.

Other formats possible (specify before .TL) are: .MR ("memo for record"), .MF ("memo for file"), .EG ("engineer's notes") and .TR (Computing Science Tech. Report).

Headings

.NH Introduction. .PP text text text	.SH Appendix I .PP text text text
1. Introduction text text text	Appendix I text text text

A Simple List

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP (terminates list)
```

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw*, Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards*, Proc. ASME **23** (1974), 23-24.

Displays

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

    and now
    for something
    completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: .DS L: left-adjust; .DS C: line-by-line center; .DS B: make block, then center.

Footnotes

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without these basic tools*
.FS
* As first shown by Tiger & Leopard
(1975).
.FE
few assemblies could be completed. They may
lack the popular appeal of the sledgehammer
```

Among the most important occupants of the workbench are the long-nosed pliers. Without these basic tools* few assemblies could be completed. They may lack the popular appeal of the sledgehammer

---

* As first shown by Tiger & Leopard (1975).

Multiple Indents

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the
indenting at this point.
.IP 3.
Another
line.
```

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting at this point.
3. Another line.

Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

.KS	not moved	.KF	may float
.KE	through text	.KE	in text

Double Column

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes
necessary for one people to dissolve the
political bonds which have connected them with
another, and to assume among the powers of the
earth the separate and equal station to which
the laws of Nature and of Nature's God entitle
them, a decent respect to the opinions of
```

The Declaration of Independence

When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their creator with certain unalienable rights, that among these are life, liberty, and the pursuit of happiness. That to secure these rights, governments are instituted among men,

Equations

A displayed equation is marked
with an equation number at the right margin
by adding an argument to the EQ line:

```
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

```
.EQ I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above .} col {. above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above  gamma } right ]
.EN
```

$$\bar{\mathbf{V}}_\nu = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad (2.2a)$$

```
.EQ L
F hat ( chi ) ~ mark = ~| del V | sup 2
.EN
.EQ L
lineup =~ {left ( {partial V} over {partial x} right )
} sup 2 + { left ( {partial V} over {partial y} right
) } sup 2 ~~~~~~ lambda -> inf
.EN
```

$$\hat{F}(\chi) = |\nabla V|^2$$

$$= \left(\frac{\partial V}{\partial x}\right)^2 + \left(\frac{\partial V}{\partial y}\right)^2 \qquad \lambda \to \infty$$

$ a dot $, $ b dotdot$, $ xi tilde times y vec$:

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$.   (with delim $$ on, see panel 3).

See also the equations in the second table, panel 8.

Some Registers You Can Change

Line length
  .nr LL 7i

Title length
  .nr LT 7i

Point size
  .nr PS 9

Vertical spacing
  .nr VS 11

Column width
  .nr CW 3i

Intercolumn spacing
  .nr GW .5i

Margins — head and foot
  .nr HM .75i
  .nr FM .75i

Paragraph indent
  .nr PI 2n

Paragraph spacing
  .nr PD 0

Page offset
  .nr PO 0.5i

Page heading
  .ds CH Appendix
    (center)
  .ds RH 7-25-76
    (right)
  .ds LH Private
    (left)

Page footer
  .ds CF Draft
  .ds LF
  .ds RF similar

Page numbers
  .nr % 3

Tables

```
.TS          (Ⓣ indicates a tab)
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (x) and linesize (n).

```
.TS          (with delim $$ on, see panel 3)
doublebox, center;
c c
l l.
NameⓉDefinition
.sp
GammaⓉ$GAMMA (z) = int sub 0 sup inf \
  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi \
  int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi \
  int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = \
  sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.TE
```

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^\infty k^{-s} \quad (\mathrm{Re}\ s>1)$

Usage

Documents with just text:
troff -ms files
With equations only:
eqn files | troff -ms
With tables only:
tbl files | troff -ms
With both tables and equations:
tbl files|eqn|troff -ms

The above generates STARE output on GCOS: replace −st with −ph for typesetter output.

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (−) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
−**o***list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N*−*M* and means pages *N* through *M;* a initial −*N* means from the beginning to page *N;* and a final *N*− means from *N* to the end.
−**n***N*	Number first generated page *N*.
−**s***N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
−**m***name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files*.
−**r***aN*	Register *a* (one-character) is set to *N*.
−**i**	Read standard input after the input files are exhausted.
−**q**	Invoke the simultaneous input-output mode of the **rd** request.

NROFF Only

−T*name* Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

−e Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

−t Direct output to the standard output instead of the phototypesetter.

−f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

−w Wait until phototypesetter is available, if currently busy.

−b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

−a Send a printable (ASCII) approximation of the results to the standard output.

−p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

−g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff **−o***4,8−10* **−T***300S* **−m***abc* *file1* *file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl *files* | **eqn** | **troff** **−t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*

1. General Explanation

2. Font and Character Size Control

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.ps ±*N*	10 point	previous	E	Point size; also \s±*N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *FNM*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also **\f***x*, **\f(***xx*, **\f***N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position 1≤*N*≤4.

3. Page Control

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.pl ±*N*	11 in	11 in	v	Page length.
.bp ±*N*	*N*=1	-	B‡,v	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	0; 26/27 in	previous	v	Page offset.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,v	Return *(upward only)* to marked vertical place.

4. Text Filling, Adjusting, and Centering

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.

5. Vertical Spacing

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,p	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 *V*s after each text output line.
.sp *N*	-	*N*=1 *V*	B,v	Space vertical distance *N in either direction.*
.sv *N*	-	*N*=1 *V*	v	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.

6. Line Length and Indenting

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.ll ±*N*	6.5 in	previous	E,m	Line length.
.in ±*N*	*N*=0	previous	B,E,m	Indent.
.ti ±*N*	-	ignored	B,E,m	Temporary indent.

7. Macros, Strings, Diversion, and Position Traps

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
.de *xx yy*	-	*.yy*=..	-	Define or redefine macro *xx;* end at call of *yy*.
.am *xx yy*	-	*.yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.
.as *xx string*	-	ignored	-	Append *string* to string *xx*.

---

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of " ' " as control character (instead of ".") suppresses the break function.

Request Form	Initial Value	If No Argument	Notes	Explanation
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	v	Change trap location.
.dt *N xx*	-	off	D,v	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.

8. Number Registers

Request Form	Initial Value	If No Argument	Notes	Explanation
.nr *R* ±*N M*		-	u	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.rr *R*	-	-	-	Remove register *R*.

9. Tabs, Leaders, and Fields

Request Form	Initial Value	If No Argument	Notes	Explanation
.ta *Nt* ...	0.8; 0.5in	none	E,m	Tab settings; *left* type, unless *t*=**R**(right), **C**(centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	.	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.

10. Input and Output Conventions and Character Translations

Request Form	Initial Value	If No Argument	Notes	Explanation
.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	-; on	on	-	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in TROFF) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in NROFF; like **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by **ul**).
.cc *c*	.	.	E	Set control character to *c*.
.c2 *c*	'	'	E	Set nobreak control character to *c*.
.tr *abcd....*	none	-	O	Translate *a* to *b*, etc. on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

13. Hyphenation.

Request Form	Initial Value	If No Argument	Notes	Explanation
.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1* ...		ignored	-	Exception words.

14. Three Part Titles.

Request Form	Initial Value	If No Argument	Notes	Explanation
.tl *'left'center'right'*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt ±*N*	6.5 in	previous	E,m	Length of title.

15. Output Line Numbering.

Request Form	Initial Value	If No Argument	Notes	Explanation
.nm ±*N M S I*		off	E	Number mode on or off, set parameters.
.nn *N*	-	*N*=1	E	Do not number next *N* lines.

16. Conditional Acceptance of Input

Request Form	Initial Value	If No Argument	Notes	Explanation
.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \{*anything*\}.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if !*c anything*		-	-	If condition *c* false, accept ***anything***.
.if *N anything*		-	u	If expression $N > 0$, accept ***anything***.
.if !*N anything*		-	u	If expression $N \leqslant 0$, accept ***anything***.
.if *'string1'string2' anything*			-	If ***string1*** identical to ***string2***, accept ***anything***.
.if !*'string1'string2' anything*			-	If ***string1*** not identical to ***string2***, accept ***anything***.
.ie *c anything*		-	u	If portion of if-else; all above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.
17. Environment Switching.				
.ev *N*	*N*=0	previous	-	Environment switched (***push down***).
18. Insertions from the Standard Input				
.rd *prompt*	-	*prompt*=BEL	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.
19. Input/Output File Switching				
.so *filename*		-	-	Switch source file ***(push down)***.
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to ***program*** (NROFF only).
20. Miscellaneous				
.mc *c N*	-	off	E,m	Set margin character *c* and separation *N*.
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=..	-	Ignore till call of *yy*.
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

- B Request normally causes a break.
- D Mode or relevant parameters associated with current diversion level.
- E Relevant parameters are a part of the current environment.
- O Must stay in effect until logical output.
- P Mode must be still or again in effect at the time of physical output.
- v,p,m,u Default scale indicator; if not specified, scale indicators are *ignored.*

Alphabetical Request and Section Number Cross Reference

ad 4	cc 10	ds 7	fc 9	ie 16	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	dt 7	fi 4	if 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	ec 10	fl 20	ig 20	lt 14	nn 15	pm 20	rs 5	ti 6	
as 7	cs 2	el 16	fp 2	in 6	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cu 10	em 7	ft 2	it 7	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	da 7	eo 10	hc 13	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	de 7	ev 17	hw 13	lg 10	ne 3	os 5	rd 18	ss 2	uf 10	
c2 10	di 7	ex 18	hy 13	li 10	nf 4	pc 14	rm 7	sv 5	ul 10	

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay the interpretation of \ )
10.1	\e	Printable version of the *current* escape character.
2.1	\´	´ (acute accent); equivalent to \(**aa**
2.1	\`	` (grave accent); equivalent to \(**ga**
2.1	\−	− Minus sign in the *current* font
7	\.	Period (dot) (see **de**)
11.1	\(space)	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	|	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.7	\"	Beginning of comment
7.3	\$*N*	Interpolate argument $1 \leq N \leq 9$
13	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	**x*, *(*xx*	Interpolate string *x* or *xx*
9.1	\a	Non-interpreted leader character
12.3	\b'*abc...*'	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\f*x*,\f(*xx*,\f*N*	Change to font named *x* or *xx*, or position *N*
11.1	\h'*N*'	Local horizontal motion; move right *N* *(negative left)*
11.3	\k*x*	Mark horizontal *input* place in register *x*
12.4	\l'*Nc*'	Horizontal line drawing function (optionally with *c*)
12.4	\L'*Nc*'	Vertical line drawing function (optionally with *c*)
8	\n*x*,\n(*xx*	Interpolate number register *x* or *xx*
12.1	\o'*abc...*'	Overstrike characters *a, b, c, ...*
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	\s*N*, \s±*N*	Point-size change function
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	\v'*N*'	Local vertical motion; move down *N* *(negative up)*
11.2	\w'*string*'	Interpolate width of *string*
5.2	\x'*N*'	Extra line-space function *(negative before, positive after)*
12.2	\z*c*	Print *c* with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	*X*	*X*, any character *not* listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	%	Current page number.
11.2	ct	Character type (set by *width* function).
7.4	dl	Width (maximum) of last completed diversion.
7.4	dn	Height (vertical size) of last completed diversion.
-	dw	Current day of the week (1-7).
-	dy	Current day of the month (1-31).
11.3	hp	Current horizontal place on *input* line.
15	ln	Output line number.
-	mo	Current month (1-12).
4.1	nl	Vertical position of last printed text base-line.
11.2	sb	Depth of string below base line (generated by *width* function).
11.2	st	Height of string above base line (generated by *width* function).
-	yr	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	.$	Number of arguments available at the current macro level.
-	.A	Set to 1 in TROFF, if **−a** option used; always 1 in NROFF.
11.1	.H	Available horizontal resolution in basic units.
-	.T	Set to 1 in NROFF, if **−T** option used; always 0 in TROFF.
11.1	.V	Available vertical resolution in basic units.
5.2	.a	Post-line extra line-space most recently utilized using **\x**'*N*'.
-	.c	Number of *lines* read from current input file.
7.4	.d	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	.f	Current font as physical quadrant (1-4).
4	.h	Text base-line high-water mark on current page or diversion.
6	.i	Current indent.
6	.l	Current line length.
4	.n	Length of text portion on previous output line.
3	.o	Current page offset.
3	.p	Current page length.
2.3	.s	Current point size.
7.5	.t	Distance to the next trap.
4.1	.u	Equal to 1 in fill mode and 0 in nofill mode.
5.1	.v	Current vertical line spacing.
11.2	.w	Width of previous character.
-	.x	Reserved version-dependent register.
-	.y	Reserved version-dependent register.
7.4	.z	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ′ (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ′ suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function **\n***R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in **\n***x*, or left-parenthesis-introduced, two-character name as in **\n(***xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions, of various typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the **−T** option (default Model 37 Teletype).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as −> (→) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling.

The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

.sp |3.2c

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)/2u

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
´	acute accent	’	close quote
`	grave accent	‘	open quote
−	minus	-	hyphen

The characters ´, `, and − may be input by \´, \`, and \− respectively or by their names (Table II). The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The

characters ´, `, and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (S) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either **\f***x*, **\f(***xx*, or **\f***N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored.* TROFF can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a **\s***N* at the desired point to set the size to *N*, or a **\s**$\pm N$ ($1 \leq N \leq 9$) to increment/decrement the size by *N*; **\s0** restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the .s register. NROFF ignores type size control.

Request Form	***Initial Value***	***If No Argument***	***Notes****	***Explanation***
.ps $\pm N$	10 point	previous	E	Point size set to $\pm N$. Alternatively imbed **\s***N* or **\s**$\pm N$. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence $+N, -N$ will work because the previous requested value is also remembered. Ignored in NROFF.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *F N M*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd *F N*	off	-	P	The characters in font *F* will be artificially emboldened by printing each one twice, separated by $N-1$ basic units. A reasonable value for *N* is 3 when the character size is in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The column heads above were printed with **.bd I 3**. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B 3**. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (*N* from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp $\pm N$	*N*=1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request **ns**.
.pn $\pm N$	*N*=1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current *left margin* is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches. See §6. The current page offset is available in the **.o** register.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space. If the distance, *D*, to the next trap position (see §7.5) is less than *N*, a forward vertical space of size *D* occurs, which will spring the trap. If there are no remaining traps on the page, *D* is the

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

				distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, *D* is the distance to the *diversion trap*, if any, or is very large.
.mk *R*	none	internal	D	Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.
.rt $\pm N$	none	internal	D,v	Return *upward only* to a marked vertical place in the current diversion. If $\pm N$ (w.r.t. current place) is given, the place is $\pm N$ from the top of the page or diversion or, if *N* is absent, to a place marked by a previous **mk**. Note that the **sp** request (§5.3) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; e. g. using the sequence **.mk** *R* ... **.sp** **|\n***R***u**.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the **ss** request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **−e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, or **!** is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using **tr** (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a **\c**. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with **\c**; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

.fi	fill on	-	B,E	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
.ad *c*	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
.ce *N*	off	*N*=1	B,E	Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function **\x**'*N*' can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	***Initial Value***	***If No Argument***	***Notes***	***Explanation***
.vs *N*	1/6in;12pts	previous	E,p	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**'*N*' (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ±*N*. *N*−1 *V*s *(blank lines)* are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line

				reached a trap position.
.sp *N*	-	*N*=1 *V*	B,v	Space vertically in *either* direction. If *N* is negative, the motion is *backward* (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see **ns**, and **rs** below).
.sv *N*	-	*N*=1 *V*	v	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see **os**). Subsequent **sv** requests will overwrite any still remembered *N*.
.os	-	-	-	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier **sv** request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with **rs**.
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like **sp 1**.

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	***Initial Value***	***If No Argument***	***Notes***	***Explanation***
.ll ±*N*	6.5 in	previous	E,m	Line length is set to ±*N*. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches.
.in ±*N*	*N*=0	previous	B,E,m	Indent is set to ±*N*. The indent is prepended to each output line.
.ti ±*N*	-	ignored	B,E,m	Temporary indent. The *next* output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a

control line beginning *.xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with *****x* and *****(xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by **\n** are interpolated.
- Strings indicated by ***** are interpolated.
- Arguments indicated by **\$** are interpolated.
- Concealed newlines indicated by \(newline) are eliminated.
- Comments indicated by **\"** are eliminated.
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH respectively (§9).
- **** is interpreted as ****.
- **\\.** is interpreted as ".".

These interpretations can be suppressed by prepending a ****. For example, since **** maps into a ****, **\\\\n** will copy as **\n** which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with **\$***N*, which interpolates the *N*th argument ($1 \leqslant N \leqslant 9$). If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx      \"begin definition
Today is \\$1 the \\$2.
..          \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

Today is Monday the 14th.

Note that the **\$** was concealed in the definition with a prepended ****. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra ****) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way

to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

*7.5. **Traps.*** Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see **it** below.

Request Form	***Initial Value***	***If No Argument***	***Notes***	***Explanation***
.de *xx yy*	-	*.yy*=..	-	Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with *.yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with "..". A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. ".." can be concealed as \\.. which will copy as \.. and be reread as "..".
.am *xx yy*	-	*.yy*=..	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.
.di *xx*	-	end	D	Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request **di** or **da** is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

.da *xx*	-	end	D	Divert, appending to *xx* (append version of **di**).
.wh *N xx*	-	-	v	Install a trap to invoke *xx* at page position *N;* a *negative N* will be interpreted with respect to the page *bottom.* Any macro previously planted at *N* is replaced by *xx.* A zero *N* refers to the *top* of a page. In the absence of *xx*, the first found trap at *N*, if any, is removed.
.ch *xx N*	-	-	v	Change the trap position for macro *xx* to be *N.* In the absence of *N*, the trap, if any, is removed.
.dt *N xx*	-	off	D,v	Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx.* Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it *N xx*	-	off	E	Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
.em *xx*	none	none	-	The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	*N+M*
\n−*x*	*x* decremented by *M*	*N−M*
\n+(*xx*	*xx* incremented by *M*	*N+M*
\n−(*xx*	*xx* decremented by *M*	*N−M*

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nr *R* ±*N M*		-	u	The number register *R* is assigned the value ±*N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M.*

.af *R c* arabic - - Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

.rr *R* - ignored - Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with **ta**. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and *W* is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode.* **\t** and **\a** always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode.*

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is **#** and the padding indicator is **^**, **#^***xxx***^***right***#** specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *Nt* ...	0.8; 0.5in	none	E,m	Set tab stops and types. *t*=**R**, right adjusting; *t*=**C**, centering; *t* absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by **+** is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All* others are ignored.

The *escape* character \ introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with **ec**, and all that has been said about the default \ becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **ﬁ**, **ﬂ**, **ﬀ**, **ﬃ**, and **ﬄ**. They may be input (even in NROFF) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in *copy mode*. No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\f***F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in NROFF (italicize in TROFF) the next *N* input text lines. Actually, switch to *underline* font, saving the current font for later restoration; *other* font changes within the span of a **ul** will take effect, but the restoration will undo the last change. Output generated by **tl** (§14) *is* affected by the font change, but does *not* decrement *N*. If *N*> 1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character . and the *no-break* control character ′ may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	′	′	E	The *nobreak* control character is set to *c*, or reset to "′".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a \! is read in *copy mode* and *transparently* output (without the initial \!); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \(newline) is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with \". The newline at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with .\".

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v**′*N*′ and **\h**′*N*′ can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in TROFF	NROFF	Horizontal Local Motion	Effect in TROFF	NROFF
\v'*N*'	Move distance *N*		\h'*N*'	Move distance *N*	
\u	½ em up	½ line up	\(space)	Unpaddable space-size space	
\d	½ em down	½ line down	\0	Digit-size space	
\r	1 em up	1 line up	\\|	1/6 em space	ignored
			\^	1/12 em space	ignored

As an example, E^2 could be generated by the sequence **E\s−2\v'−0.4m'2\v'0.4m'\s+2**; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function **\w'*string*'** generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, **.ti −\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string "**1.** ".

The width function also sets three number registers. The registers **st** and **sb** are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is **\n(stu−\n(sbu**. In TROFF the number register **ct** is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like **e**); 1 means that at least one character has a descender (like **y**); 2 means that at least one character is tall (like **H**); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence **\k*x*** will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\kx*word*\h'|\nxu+2u'*word*** will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function **\o'*string*'**. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, **\o'e\''** produces é, and **\o'\(mo\(sl'** produces ∉.

12.2. Zero-width characters. The function **\z*c*** will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** will produce ⊕, and **\(br\z\(rn\(ul\(br** will produce the smallest possible constructed box ▯.

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (⎧⎩⎫⎭⎨⎬│⌊⌋⌈⌉) that can be combined into various bracket styles. The function **\b'*string*'** may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, **\b'\(lc\(lf'E\|\b'\(rc\(rf'\x'−0.5m'\x'0.5m'** produces [E].

12.4. Line drawing. The function **\l'*Nc*'** will draw a string of repeated *c*'s towards the right for a distance *N*. (**\l** is \(lower case L). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a **\&**. If *c* is not specified, the _ (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule _, underrule _, and root-en ‾, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\l'|0\(rn'\l'|0\(ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and |words in a box|.

The function **\L**′ *Nc*′ will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (**\(br**); the other suitable character is the *bold vertical* | (**\(bv**). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp −1        \"compensate for next automatic base-line spacing
.nf           \"avoid possibly overflowing word buffer
\h'−.5n'\L'|\\nau−1'\l'\\n(.lu+1n\(ul'\L'−|\\nau+1'\l'|0u−.5n\(ul'   \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using **.mk a**) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hy*N*	on,N=1	on,N=1	E	Automatic hyphenation is turned on for $N \geqslant 1$, or off for N=0. If N=2, *last* lines (ones that will cause a trap) are not hyphenated. For N=4 and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. N=14 will invoke all three restrictions.
.hc *c*	\%	\%	E	Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output.
.hw *word1* ...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are

implied; i. e. *dig−it* implies *dig−its.* This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tl *'left'center'right'*		-	-	The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially **%**) is found within any of the fields it is replaced by the current page number having the format assigned to register **%**. Any character may be used as the string delimiter.
.pc *c*	%	off	-	The page number character is set to *c*, or removed. The page-number register remains **%**.
.lt $\pm N$	6.5 in	previous	E,m	Length of title set to $\pm N$. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are
3 thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with
6 **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear
9 as blank number fields).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nm $\pm N\ M\ S\ I$		off	E	Line number mode. If $\pm N$ is given, line numbering is turned on, and the next output line numbered is numbered $\pm N$. Default values are $M=1$, $S=1$, and $I=0$. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**.
.nn *N*	-	N=1	E	The next *N* text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with $M=3$: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed
12 in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w'0000'u**) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with
15 $M=5$, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, ! signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if *c anything*		-	-	If condition *c* true, accept *anything* as input; in multi-line case use \{*anything*\}.
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*		-	u	If expression $N > 0$, accept *anything*.
.if !*N anything*		-	u	If expression $N \leq 0$, accept *anything*.
.if *'string1'string2' anything*			-	If *string1* identical to *string2*, accept *anything*.
.if !*'string1'string2' anything*			-	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*		-	u	If portion of if-else; all above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \{ and the last line must end with a right delimiter \}.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp 0.5i
.tl 'Page %'''
'sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters,

number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with **.ev** rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with **rd**, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.**rd** *prompt*	-	*prompt*=BEL	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, *prompt* (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments may be placed after *prompt*.
.**ex**	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option **−q** will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.so *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. **so**'s may be nested.
.**nx** *filename*		end-of-file	-	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.
.**pi** *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

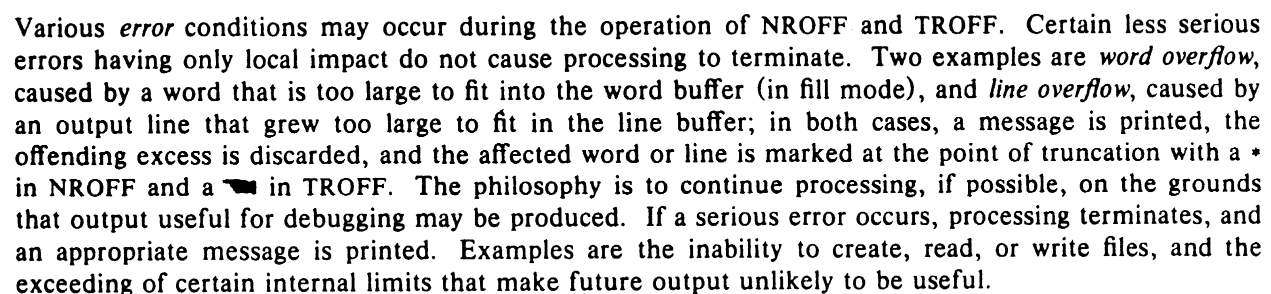
20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.mc *c N*	-	off	E,m	Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by **tl**). If the output line is too-long (as can happen in nofill mode) the character will

				be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal.
.ig *yy*	-	*.yy*=..	-	Ignore input lines. **ig** behaves exactly like **de** (§7) except that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.



Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in NROFF and a ☞ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh −1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd              \"header
.if t .tl ´\(rn´´\(rn´  \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i−1         \"tl base at 0.5i
.tl ´´− % −´´       \"centered page number
.ps                 \"restore size
.ft                 \"restore font
.vs \}              \"restore vs
'sp |1.0i           \"space to 1.0i
.ns                 \"turn on no-space mode
..
.de fo              \"footer
.ps 10              \"set footer/header size
.ft R               \"set font
.vs 12p             \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu−0.5i−1 \"tl base 0.5i up
.tl ´´− % −´´ \} \"first page number
'bp
..
.wh 0 hd
.wh −1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The sp's refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as

---

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s     \"current size
.ps
.nr s2 \\n(.s     \"previous size
. ---             \"rest of footer
..
.de hd
. ---             \"header stuff
.ps \\n(s2        \"restore previous size
.ps \\n(s1        \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn            \"bottom number
.tl ''- % -''     \"centered page number
..
.wh −0.5i−1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg            \"paragraph
.br               \"break
.ft R             \"force font,
.ps 10            \"size,
.vs 12p           \"spacing,
.in 0             \"and indent
.sp 0.4           \"prespace
.ne 1+\\n(.Vu \"want more than 1 line
.ti 0.2i          \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc            \"section
. ---             \"force font, etc.
.sp 0.4           \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1         \"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp            \"labeled paragraph
.pg
.in 0.5i          \"paragraph indent
.ta 0.2i 0.5i     \"label, paragraph
.ti 0
\t\\$1\t\c        \"flow into paragraph
..
```

The intended usage is "**.lp** *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\c** so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd          \"header
. ---
.nr cl 0 1      \"init column count
.mk             \"mark top of text
..
.de fo          \"footer
.ie \\n+(cl<2 \{\
.po +3.4i       \"next column; 3.1+0.3
.rt             \"back to mark
.ns \}          \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
. ---
'bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another **.mk** would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial **.fn** and a terminal **.ef**:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd          \"header
. ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..
.de fo          \"footer
.nr dn 0        \"zero last diversion size
.if \\nx \{\
.ev 1           \"expand footnotes in ev1
.nf             \"retain vertical size
.FN             \"footnotes
.rm FN          \"delete it
.if "\\n(.z"fy" .di \"end overflow diversion
.nr x 0         \"disable fx
.ev \}          \"pop environment
. ---
'bp
..
.de fx          \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn          \"start footnote
.da FN          \"divert (append) footnote
.ev 1           \"in environment 1
.if \\n+x=1 .fs \"if first, include separator
.fi             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr z \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
                \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v \"it didn't fit
..
.de fs          \"separator
\l'1i'          \"1 inch rule
.br
..
.de fz          \"get leftover footnote
.fn
.nf             \"retain vertical size
.fy             \"where fx put it
.ef
..
.nr b 1.0i      \"bottom margin size
.wh 0 hd        \"header trap
.wh 12i fo      \"footer trap, temp position
.wh -\\nbu fx   \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**. **y** is then decremented by the size of the

footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redivert the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en          \"end-macro
\c
'bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

Font Style Examples

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available from that company.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ~ / < > { } # @ + − = ∗
α β γ δ ϵ ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ ϕ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ∼ ≃ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ 🔔 | ○ ⎧ ⎩ ⎫ ⎭ ⎨ ⎬ │ ⎣ ⎦ ⎡ ⎤ │

Table II

Input Naming Conventions for ′, ‵,and − and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

Char	Input Name	Character Name
’	′	close quote
‘	‵	open quote
—	\(em	3/4 Em dash
-	−	hyphen or
-	\(hy	hyphen
−	\−	current font minus
•	\(bu	bullet
□	\(sq	square
_	\(ru	rule
¼	\(14	1/4
½	\(12	1/2
¾	\(34	3/4

Char	Input Name	Character Name
fi	\(fi	fi
fl	\(fl	fl
ff	\(ff	ff
ffi	\(Fi	ffi
ffl	\(Fl	ffl
°	\(de	degree
†	\(dg	dagger
′	\(fm	foot mark
¢	\(ct	cent sign
®	\(rg	registered
©	\(co	copyright

Non-ASCII characters and ′, ‵, _, +, −, =, and ∗ on the special font.

The ASCII characters @, #, ", ′, ‵, <, >, \, {, }, ˜, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	Input Name	Character Name
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
∗	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ε	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota

Char	Input Name	Character Name
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
ς	\(ts	terminal sigma
τ	\(*t	tau
υ	\(*u	upsilon
φ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega
Α	\(*A	Alpha†
Β	\(*B	Beta†

Char	*Input Name*	*Character Name*
Γ	\(*G	Gamma
Δ	\(*D	Delta
Ε	\(*E	Epsilon†
Ζ	\(*Z	Zeta†
Η	\(*Y	Eta†
Θ	\(*H	Theta
Ι	\(*I	Iota†
Κ	\(*K	Kappa†
Λ	\(*L	Lambda
Μ	\(*M	Mu†
Ν	\(*N	Nu†
Ξ	\(*C	Xi
Ο	\(*O	Omicron†
Π	\(*P	Pi
Ρ	\(*R	Rho†
Σ	\(*S	Sigma
Τ	\(*T	Tau†
Υ	\(*U	Upsilon
Φ	\(*F	Phi
Χ	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
‾	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
∼	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of

Char	*Input Name*	*Character Name*
|	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand
☜	\(lh	left hand
Ⓐ	\(bs	Bell System logo
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
┃	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

May 15, 1977

Summary of Changes to N/TROFF Since October 1976 Manual

Options

-h — (Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-z — Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

.ad c — The adjustment type indicator "c" may now also be a number previously obtained from the ".j" register (see below).

.so name — The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text — Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N — forces font "F" to be in size N. N may have the form N, +N, or -N. For example,

```
.fz 3 -2
```

will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,

```
.fz S F N
```

may be used to specify the size treatment of special characters during font F. For example,

```
.fz 3 -3
.fz S 3 -0
```

will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers.

.k — Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.

.j — Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.

.P — Read-only. 1 if the current page is being printed, and zero otherwise.

.L — Read-only. Contains the current line-spacing parameter ("ls").

c. — General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX† and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '−ms' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '−ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

2. Point sizes and line spacing
3. Fonts and special characters
4. Indents and line length
5. Tabs
6. Local motions: Drawing lines and characters
7. Strings
8. Introduction to macros
9. Titles, pages and numbering
10. Number registers and arithmetic
11. Macros with arguments
12. Conditionals
13. Environments
14. Diversions

Appendix: Typesetter character set

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. (Readers who use **roff** will find the approach familiar.) For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area = πr^2

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are ½ inch. There are 15 point sizes, listed below.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after .ps is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, **troff** reverts to the previous size, whatever it was. **troff** begins with point size 10, which is usually fine. This document is in 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

UNIX runs on a PDP-11/45

type

```
\s8UNIX\s10 runs on a \s8PDP-\s1011/45
```

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s−2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command **.sp** is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that's more or less than you want, so **.sp** can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever

.vs is set to (this can also be made explicit with **.sp** 2v); **troff** also understands decimal fractions in most places, so

    .sp 1.5i

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

    .ft B

and for italics,

    .ft I

To return to roman, use .ft **R**; to return to the previous font, whatever it was, use either .ft **P** or just .ft. The 'underline' command

    .ul

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command **\f**:

bold*face* text

is produced by

    \fBbold\fIface\fR text

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra **\fP** commands, like this:

    \fBbold\fP\fIface\fP\fR text\fP

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of **.ps** and **.vs** when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

    .fp 3 H

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, \f3 and .ft~3 mean 'whatever font is mounted at position 3', and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

    ¼ + ½ = ¾

is produced by

    \(14 + \(12 = \(34

In particular, greek letters are all of the form \(*–, where – is an upper or lower case roman letter reminiscent of the greek. Thus to get

$$\Sigma(\alpha\times\beta)\rightarrow\infty$$

in bare **troff** we have to type

    \(*S(\(*a\(mu\(*b) \(-> \(if

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	$\times$
\(*b	β
)	)
\(->	$\rightarrow$
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] the same effect can be achieved with the input

SIGMA (alpha times beta) −> inf

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

.tr \(mi\(em

is perfectly clear, meaning

.tr −—

that is, to translate − into —.

Some characters are automatically translated into others: grave ` and acute ´ accents (apostrophes) become open and close single quotes '‒'; the combination of "..." is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit − sign, use \-. To get a backslash printed, use \e.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the .ll command, as in

.ll 6i

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

.po 0

sets the offset as far to the left as it will go.

The indent command .in causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll −0.3i
text to be set into a block
.ll +0.3i
.in −0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of '+' and '−' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With .in, .ll and .po, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command .ti. For example, all paragraphs in this memo effectively begin with the command

.ti 3

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in .ti 2.5m.

Lines can also be indented negatively if the indent is already positive:

.ti −0.3i

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a .ti command:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a '\s36P\s0'), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the .ta command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

1	2	3
40	50	60
700	800	900

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru        (\(ru is "_")
Name tab Age tab
```

produces

Name ________________ Age __________

To reset the tab replacement character to a blank, use .tc with no argument. (Lines can also be drawn with the \l command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions \u and \d. To go back up the page half a point-size, insert a \u at the desired place; to go down, insert a \d. (\u and \d should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with \s−2...\s0. Since \u and \d refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by \u and \d isn't the right amount. The \v command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in '(amount)'. For example, to move the 'P' down, we used

```
.in +0.6i        (move paragraph in)
.ll −0.3i        (shorten lines)
.ti −0.3i        (move P back)
\v'2'\s36P\s0\v'−2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus \v'−2' causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'−0.5m'
```

and so on are all legal. Notice that the scale specifier i or p or m goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus \v, like \u and \d, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — \h is quite analogous to \v, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'−0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

>\h'−0.3m'>

to produce >>.

Frequently **\h** is used with the 'width function' **\w** to generate motions equal to the width of some character string. The construction

\w'thing'

is a number equal to the width of 'thing' in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

\h'\w'x'u'

As we mentioned above, the default scale factor for all horizontal dimensions is **m**, ems, so here we must have the **u** for machine units, or the motion produced will be far too large. **troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like **.sp**, were done by overstriking with a slight offset. The commands for **.sp** are

.sp\h'−\w'.sp'u'\h'1u'.sp

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen **\0**, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, **\|**, which is half that width, **\^**, which is one quarter of the width of a space, and **\&**, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command \o, used like

\o'set of characters'

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique

which makes

système téléphonique

The accents are **\(ga** and **\(aa**, or **\`** and **\´**; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, **\z**, the zero-motion command. **\zx** suppresses the normal horizontal motion after printing the single character **x**, so another character can be laid on top of it. Although sizes can be changed within \o, it centers the characters on the widest, and there can be no horizontal or vertical motions, so \z may be the only way to get what you want:

□

is produced by

.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq

The **.sp** is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or ;

can be constructed with a big comma and a big period above it:

\s+6\z,\v'−0.25m'.\v'0.25m'\s0

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function **\b**, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

{[x]}

by typing in only this:

.sp
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \b'\(rc\(rf' \b'\(rt\(rk\(rb'

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. \l'1i' draws a line one inch long, like this: ______________. The length can be followed by the character to use if the _ isn't appropriate; \l'0.5i.' draws a half-inch line of dots: .... The construction \L is entirely analogous, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing \o"e\´" for each é would be a

great nuisance.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command **.ds**. The line

```
.ds e \o"e\'"
```

defines the string **e** to have the value **\o"e\'"**

String names may be either one or two characters long, and are referred to by ***x** for one character names or ***(xy** for two character names. Thus to get téléphone, given the definition of the string **e** as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "      text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way we tell **troff** what **.PP** means is to *define* it with the **.de** command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used '**.PP**' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of .PP has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of **.PP** to something like

```
.de PP        \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used **.PP**.

\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS      \" start indented block
.sp
.nf
.in +0.3i
..
.de BE      \" end indented block
.sp
.fi
.in −0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands **.BS** and **.BE**, and it will come out as it did above. Notice that we indented by **.in +0.3i** instead of **.in 0.3i**. This way we can nest our uses of **.BS** and **BE** to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of **.BS** and **.BE**, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

```
~~~~left top          center top          right top~~~~
```

In **roff**, one can say

```
.he ′left top′center top′right top′
.fo ′left bottom′center bottom′right bottom′
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro **.NP** (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
′bp
′sp 0.5i
.tl ′left top′center top′right top′
′sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command **′bp**, which causes a skip to top-of-page (we'll explain the **′** shortly). Then we space down half an inch, print the title (the use of **.tl** should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for **.NP** at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command **.wh**:

```
.wh −1i NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '−1i' means 'one inch from the bottom'.

The **.wh** command appears in the input outside the definition of **.NP**; typically the input would be

```
.de NP
...
..
.wh −1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the **.NP** macro is activated. (In the jargon, the **.wh** command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) **.NP** causes a skip to the top of the next page (that's what the **′bp** was for), then prints the title with the appropriate margins.

Why **′bp** and **′sp** instead of **.bp** and **.sp**? The answer is that **.sp** and **.bp**, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used **.sp** or **.bp** in the **.NP** macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using **′** instead of **.** for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp  .br  .ce  .fi  .nf  .sp  .in  .ti
```

All others cause *no* break, regardless of whether

you use a . or a '. If you really need a break, add a .br command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the .lt command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change .NP to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R        \" set title font to roman
.ps 10       \" and size to 10 point
.lt 6i       \" and length to 6 inches
.tl 'left'center'right'
.ps          \" revert to previous size
.ft P        \" and to previous font
'sp 0.3i
..
```

This version of .NP does *not* work if the fields in the .tl command contain size or font changes. To cope with that requires **troff's** 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify .NP so it does some processing before the 'bp command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character **%** in the **.tl** line at the position where you want the number to appear. For example

```
.tl ''- % -''
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either **.bp n**, which immediately starts a new page numbered **n**, or with **.pn n**, which sets the page number for the next page but doesn't cause a skip to the new page. Again, **.bp +n** sets the page number to **n** more than its current value; **.bp** means **.bp +1**.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the **.nr** command, and are referenced anywhere by **\nx** (one character name) or **\n(xy** (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them **%** for the current page number; **nl** for the current vertical position on the page; **dy, mo** and **yr** for the current day, month and year; and **.s** and **.f** for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

As an example of the use of number registers, in the **−ms** macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro .PP is defined (roughly) as follows:

```
.de PP
.ps \\n(PS      \" reset size
.vs \\n(VSp     \" spacing
.ft R           \" font
.sp 0.5v        \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used*, we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes

is only needed for **\n**, *****, **\$** (which we haven't come to yet), and \ itself. Things like **\s**, **\f**, **\h**, **\v**, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS−2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*−4+3/13
```

becomes '−1'. Number registers can occur anywhere in an expression, and so can scale indicators like **p**, **i**, **m**, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so 1i/2u evaluates to 0.5i correctly.

The scale indicator **u** often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like **.ll** are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a **.nr** command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the **u** on the .ll command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro .SM that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of .SM is

```
.de SM
\s−2\\$1\s+2
..
```

Within a macro definition, the symbol **\\$n** refers to the **n**th argument that the macro was called with. Thus **\\$1** is the string to be placed in a smaller point size when .SM is called.

As a slightly more complicated version, the following definition of .SM permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s−2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register **.$**.

The following macro **.BD** is the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'−\w'\\$1'u+1u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The **\&** is there in case the argument begins with a period.

Two backslashes are needed with the **\\$n** commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .SH macro:

```
.nr SH 0        \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1   \" increment number
.ps \\n(PS−1      \" decrease PS
\\n(SH. \\$1      \" number. title
.ps \\n(PS        \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used **\\n(SH** instead of **\n(SH** and **\\n(PS** instead of **\n(PS**. If we had used **\n(SH**, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using **\\n(PS**, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .SH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i       \" first section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. **troff** also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with !; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with .if. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are t and n, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an .if:

```
.if 'string1'string2' stuff
```

does 'stuff' if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *****, arguments with **\$**, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command .ev n shifts to environment n; n must be 0, 1 or 2. The command .ev with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro .NP to process titles in environment 1 like this:

```
.de NP
.ev 1        \" shift to new environment
.lt 6i       \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev          \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the .NP macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command .di xy begins a diversion — all subsequent output is collected into the macro **xy** until the command .di with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register **dn**.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands .KS and .KE will not be split across a page boundary (as for a figure or table). Clearly, when a .KS is encountered, we have to begin diverting the output so we can find out how big it is. Then when a .KE is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS    \" start keep
.br       \" start fresh line
.ev 1     \" collect in new environment
.fi       \" make it filled text
.di XX    \" collect in XX
..

.de KE    \" end keep
.br       \" get last partial line
.di       \" end diversion
.if \\n(dn>=\\n(.t .bp  \" bp if doesn't fit
.nf       \" bring it back in no-fill
.XX       \" text
.ev       \" return to normal environment
..
```

Recall that number register **nl** is the current

position on the output page. Since output was being diverted, this remains at its value when the diversion started. **dn** is the amount of text in the diversion; .t (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the **.if** is satisfied, and a **.bp** is issued. In either case, the diverted output is then brought back with **.XX**. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of **troff**, for his repeated patient explanations of fine points, and for his continuing willingness to adapt **troff** to make other uses easier. I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories Computing Science Technical Report 54, 1976.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition),* Bell Laboratories Computing Science Technical Report 17, 1977.

[3] M. E. Lesk, *TBL — A Program to Format Tables,* Bell Laboratories Computing Science Technical Report 49, 1976.

[4] M. E. Lesk, *Typing Documents on UNIX,* Bell Laboratories, 1978.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros,* Bell Laboratories internal memorandum.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ﬀ	\(ff	ﬁ	\(fi	ﬂ	\(fl	ﬃ	\(Fi	ﬄ	\(Fl
_	\(ru	—	\(em	¼	\(14	½	\(12	¾	\(34
©	\(co	°	\(de	†	\(dg	′	\(fm	¢	\(ct
®	\(rg	•	\(bu	□	\(sq	-	\(hy		

(In bold, \(sq is ■.)

The following are special-font characters:

<table>
<tr><td>+</td><td>\(pl</td><td>−</td><td>\(mi</td><td>×</td><td>\(mu</td><td>÷</td><td>\(di</td></tr>
<tr><td>=</td><td>\(eq</td><td>≡</td><td>\(==</td><td>≥</td><td>\(>=</td><td>≤</td><td>\(<=</td></tr>
<tr><td>≠</td><td>\(!=</td><td>±</td><td>\(+-</td><td>¬</td><td>\(no</td><td>/</td><td>\(sl</td></tr>
<tr><td>~</td><td>\(ap</td><td>≃</td><td>\(~=</td><td>∝</td><td>\(pt</td><td>∇</td><td>\(gr</td></tr>
<tr><td>→</td><td>\(-></td><td>←</td><td>\(<-</td><td>↑</td><td>\(ua</td><td>↓</td><td>\(da</td></tr>
<tr><td>∫</td><td>\(is</td><td>∂</td><td>\(pd</td><td>∞</td><td>\(if</td><td>√</td><td>\(sr</td></tr>
<tr><td>⊂</td><td>\(sb</td><td>⊃</td><td>\(sp</td><td>∪</td><td>\(cu</td><td>∩</td><td>\(ca</td></tr>
<tr><td>⊆</td><td>\(ib</td><td>⊇</td><td>\(ip</td><td>∈</td><td>\(mo</td><td>∅</td><td>\(es</td></tr>
<tr><td>´</td><td>\(aa</td><td>`</td><td>\(ga</td><td>○</td><td>\(ci</td><td>◉</td><td>\(bs</td></tr>
<tr><td>§</td><td>\(sc</td><td>‡</td><td>\(dd</td><td>☜</td><td>\(lh</td><td>☞</td><td>\(rh</td></tr>
<tr><td>⎧</td><td>\(lt</td><td>⎫</td><td>\(rt</td><td>⎡</td><td>\(lc</td><td>⎤</td><td>\(rc</td></tr>
<tr><td>⎩</td><td>\(lb</td><td>⎭</td><td>\(rb</td><td>⎣</td><td>\(lf</td><td>⎦</td><td>\(rf</td></tr>
<tr><td>⎨</td><td>\(lk</td><td>⎬</td><td>\(rk</td><td>|</td><td>\(bv</td><td>ς</td><td>\(ts</td></tr>
<tr><td>|</td><td>\(br</td><td>|</td><td>\(or</td><td>_</td><td>\(ul</td><td>‾</td><td>\(rn</td></tr>
<tr><td>*</td><td>\(**</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\'	`	\`	−	\-	_	_

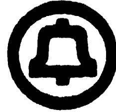
These characters exist only on the special font, but they do not have four-character names:

" { } < > ˜ ˆ \ # @

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω

A B G D E Z Y H I K L M N C O P R S T U F X Q W
A B Γ Δ E Z H Θ I K Λ M N Ξ O Π P Σ T Υ Φ X Ψ Ω



PWB/MM
Programmer's Workbench Memorandum Macros

D. W. Smith
J. R. Mashey

October 1977

Bell Telephone Laboratories, Incorporated

·PWB/MM
Programmer's Workbench Memorandum Macros

CONTENTS

1. INTRODUCTION 1
   - 1.1 Purpose 1
   - 1.2 Conventions 1
   - 1.3 Overall Structure of a Document 2
   - 1.4 Definitions 2
   - 1.5 Prerequisites and Further Reading 3
2. INVOKING THE MACROS 3
   - 2.1 The mm Command 3
   - 2.2 The -mm Flag 3
   - 2.3 Typical Command Lines 4
   - 2.4 Parameters that Can Be Set from the Command Line 5
   - 2.5 Omission of -mm 6
3. FORMATTING CONCEPTS 6
   - 3.1 Basic Terms 6
   - 3.2 Arguments and Double Quotes 7
   - 3.3 Unpaddable Spaces 7
   - 3.4 Hyphenation 7
   - 3.5 Tabs 8
   - 3.6 Special Use of the BEL Character 8
   - 3.7 Bullets 8
   - 3.8 Dashes, Minus Signs, and Hyphens 8
   - 3.9 Use of Formatter Requests 9
4. PARAGRAPHS AND HEADINGS c
   - 4.1 Paragraphs 9
   - 4.2 Numbered Headings 9
   - 4.3 Unnumbered Headings 12
   - 4.4 Headings and the Table of Contents 12
   - 4.5 First-Level Headings and the Page Numbering Style 12
   - 4.6 User Exit Macros • 13
   - 4.7 Hints for Large Documents 14
5. LISTS 14
   - 5.1 Basic Approach 14
   - 5.2 Sample Nested Lists 14
   - 5.3 Basic List Macros 15
   - 5.4 List-Begin Macro and Customized Lists • 18
6. MEMORANDUM AND RELEASED PAPER STYLES 19
   - 6.1 Title 20
   - 6.2 Author(s) 20
   - 6.3 TM Number(s) 20
   - 6.4 Abstract 21
   - 6.5 Other Keywords 21
   - 6.6 Memorandum Types 21
   - 6.7 Date and Format Changes 22
   - 6.8 Released-Paper Style 22
   - 6.9 Order of Invocation of "Beginning" Macros 22
   - 6.10 Example 23
   - 6.11 Macros for the End of a Memorandum 23
   - 6.12 Forcing a One-Page Letter 24

7. DISPLAYS 25
7.1 Static Displays 25
7.2 Floating Displays 25
7.3 Tables 26
7.4 Equations 26
7.5 Figure, Table, and Equation Captions 26
7.6 Blocks of Filled Text 27

8. FOOTNOTES 27
8.1 Automatic Numbering of Footnotes 27
8.2 Delimiting Footnote Text 27
8.3 Format of Footnote Text • 28
8.4 Spacing between Footnote Entries 29

9. PAGE HEADERS AND FOOTERS 29
9.1 Default Headers and Footers 29
9.2 Page Header 29
9.3 Even-Page Header 29
9.4 Odd-Page Header 30
9.5 Page Footer 30
9.6 Even-Page Footer 30
9.7 Odd-Page Footer 30
9.8 Footer on the First Page 30
9.9 Default Header and Footer with "Section-Page" Numbering 30
9.10 Use of Strings and Registers in Header and Footer Macros • 30
9.11 Header and Footer Example • 31
9.12 Generalized Top-of-Page Processing • 31
9.13 Generalized Bottom-of-Page Processing 31

10. TABLE OF CONTENTS AND COVER SHEET 31
10.1 Table of Contents 32
10.2 Cover Sheet 33

11. MISCELLANEOUS FEATURES 33
11.1 Bold, Italic, and Roman 33
11.2 Justification of Right Margin 33
11.3 SCCS Release Identification 34
11.4 Two-Column Output 34
11.5 Column Headings for Two-Column Output • 34
11.6 Vertical Spacing 34
11.7 Skipping Pages 35
11.8 Setting Point Size and Vertical Spacing 35

12. ERRORS AND DEBUGGING 35
12.1 Error Terminations 35
12.2 Disappearance of Output 36

13. EXTENDING AND MODIFYING THE MACROS • 36
13.1 Naming Conventions 36
13.2 Sample Extensions 37

14. CONCLUSION 38
References 39
Appendix A: DEFINITIONS OF LIST MACROS • 41
Appendix B: USER-DEFINED LIST STRUCTURES • 42
Appendix C: SAMPLE FOOTNOTES 44
Appendix D: SAMPLE LETTER 46
Appendix E: ERROR MESSAGES 50
Appendix F: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS 52

PWB/MM—Programmer's Workbench Memorandum Macros

D. W. Smith

Bell Laboratories
Piscataway, New Jersey 08854

J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Purpose

This memorandum is the user's guide and reference manual for PWB/MM (or just **-mm**), a general-purpose package of text formatting macros for use with the UNIX* text formatters *nroff* [9] and *troff* [9]. The purpose of PWB/MM is to provide to the users of PWB/UNIX a unified, consistent, and flexible tool for producing many common types of documents. Although PWB/UNIX provides other macro packages for various *specialized* formats, PWB/MM has become the standard, general-purpose macro package for most documents.

PWB/MM can be used to produce:

- Letters.
- Reports.
- Technical Memoranda.
- Released Papers.
- Manuals.
- Books.
- etc.

The uses of PWB/MM range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc.

1.2 Conventions

Each section of this memorandum explains a single facility of PWB/MM. In general, the earlier a section occurs, the more necessary it is for most users. Some of the later sections can be completely ignored if PWB/MM defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until there is enough information to obtain the desired format, then skimming the rest of it, because some details may be of use to just a few people.

Numbers enclosed in curly brackets ({}) refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (•) at the end of the section heading.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

A reference of the form *name*(N) points to page *name* in section *N* of the *PWB/UNIX User's Manual* [1].

The examples of *output* in this manual are as produced by *troff*; *nroff* output would, of course, look somewhat different (Appendix D shows *both* the *nroff* and *troff* output for a simple letter). In those

* UNIX is a Trademark of Bell Laboratories.

cases in which the behavior of the two formatters is truly different, the *nroff* action is described first, with the *troff* action following in parentheses. For example:

The title is underlined (bold).

means that the title is underlined in *nroff* and bold in *troff*.

1.3 Overall Structure of a Document

The input for a document that is to be formatted with PWB/MM possesses four major segments, any of which may be omitted; if present, they *must* occur in the following order:

- *Parameter-setting*—This segment sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers [9], and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but only performs the setup for the rest of the document.
- *Beginning*—This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body*—This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of *headings* up to seven levels deep [4]. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Six additional levels of subordination are provided by a set of *list* macros for automatic numbering, alphabetic sequencing, and "marking" of list items [5]. The body may also contain various types of displays, tables, figures, and footnotes [7, 8].
- *Ending*—This segment contains those items that occur once only, at the end of a document. Included here are signature(s) and lists of notations (e.g., "copy to" lists) [6.12]. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document [10].

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

1.4 Definitions

The term *formatter* refers to either of the text-formatting programs *nroff* and *troff*.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly [3.9], this document contains references to some of them. Full details are given in [9]. For example, the request:

.sp

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. PWB/MM supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by *requests* and *macros*. A string can be given a value via the .ds (define string) request, and its value can be obtained by referencing its name, preceded by "\•" (for 1-character names) or "\•(" (for 2-character names). For instance, the string *DT* in PWB/MM normally contains the current date, so that the *input* line:

Today is \•(DT.

may result in the following *output:*

Today is October 31, 1977.

The current date can be replaced, e.g.:

.ds DT 01/01/76

or by invoking a macro designed for that purpose {6.7.1}.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A register can be given a value using a .nr request, and be referenced by preceding its name by "\n" (for 1-character names) or "\n(" (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

.nr d 1+\n(dd

See {13.1} regarding naming conventions for requests, macros, strings, and number registers.

1.5 Prerequisites and Further Reading

1.5.1 Prerequisites. We assume familiarity with UNIX at the level given in [3] and [4]. Some familiarity with the request summary in [9] is helpful.

1.5.2 Further Reading. [9] provides detailed descriptions of formatter capabilities, while [5] provides a general overview. See [6] (and possibly [7]) for instructions on formatting mathematical expressions. See *tbl*(I) and [11] for instructions on formatting tabular data.

Examples of formatted documents and of their respective input, as well as a quick reference to the material in this manual are given in [8].

2. INVOKING THE MACROS

This section tells how to access PWB/MM, shows PWB/UNIX command lines appropriate for various output devices, and describes command-line flags for PWB/MM. Note that file names, program names, and typical command sequences apply only to PWB/UNIX; different names and command lines may have to be used on other systems.

2.1 The mm Command

The *mm*(I) command can be used to print documents using *nroff* and PWB/MM; this command invokes *nroff* with the -mm flag {2.2}. It has options to specify preprocessing by *tbl*(I) and/or by *neqn*(I), and for postprocessing by various output filters. Any arguments or flags that are not recognized by *mm*(I), e.g. -rC3, are passed to *nroff* or to PWB/MM, as appropriate. The options, which can occur in any order but *must* appear before the file names, are:

-e	*neqn*(I) is to be invoked.
-t	*tbl*(I) is to be invoked.
-c	*col*(I) is to be invoked.
-12	need 12-pitch mode. Be sure that the pitch switch on the terminal is set to 12.
-300	output is to a DASI300 terminal. This is the *default* terminal type.
-hp	output is to a HP264x.
-450	output is to a DASI450.
-tn	output is to a GE TermiNet 300.
-tn300	output is to a GE TermiNet 300.
-ti	output is to a Texas Instrument 700 series terminal.
-37	output is to a TELETYPE® Model 37.

2.2 The -mm Flag

The PWB/MM package can also be invoked by including the **-mm** flag as an argument to the formatter. It causes the file /usr/lib/tmac.m to be read and processed before any other files. This action defines the PWB/MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in [2.4] and in [9]).

- Text without tables or equations:

  mm [options] filename ...
  or **nroff** [options] **-mm** filename ...
  or **troff** [options] **-mm** filename ...

- Text with tables:

  mm -t [options] filename ...
  or **tbl** filename ... | **nroff** [options] **-mm -**
  or **tbl** filename ... | **troff** [options] **-mm -**

- Text with equations:

  mm -e [options] filename ...
  or **neqn** filename ... | **nroff** [options] **-mm -**
  or **eqn** filename ... | **troff** [options] **-mm -**

- Text with both tables and equations:

  mm -t -e [options] filename ...
  or **tbl** filename ... | **neqn** | **nroff** [options] **-mm -**
  or **tbl** filename ... | **eqn** | **troff** [options] **-mm -**

When formatting a document with *nroff*, the output should normally be processed for a specific type of terminal, because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See [2.4] as well as *gsi*(I), *hp*(I), *col*(I), and *terminals*(VII) for further information.

- DASI300 (GSI300/DTC300) in 10-pitch, 6 lines/inch mode and a line length of 65 characters:

  mm filename ...
  or **nroff -T300 -h -mm** filename ...

- DASI300 (GSI300/DTC300) in 12-pitch, 6 lines/inch mode and a line length of 80—rather than 65—characters:

  mm -12 filename ...
  or **nroff -T300-12 -rW80 -rO3 -h -mm** filename ...

  or, equivalently (and more succinctly):

  nroff -T300-12 -rT1 -h -mm filename ...

- DASI450 in 10-pitch, 6 lines/inch mode:

  mm -450 filename ...
  or **nroff -T450 -h -mm** filename ...

- DASI450 in 12-pitch, 6 lines/inch mode:

  mm -450 -12 filename ...
  or **nroff -T450-12 -rW80 -rO3 -h -mm** filename ...
  or **nroff -T450-12 -rT1 -h -mm** filename ...

- Hewlett-Packard HP264x CRT family:

  mm -hp filename ...
  or **nroff -h -mm** filename ... | **hp**

- Any terminal incapable of reverse paper motion (GE TermiNet, Texas Instruments 700 series, etc.)

  mm -tn filename ...
  or **nroff -mm** filename ... | **col**

- Versatec printer (see *vp*(I) for additional details):

```
    vp [vp-options] "mm -rT2 -c filename ..."
or  vp [vp-options] "nroff -rT2 -mm filename ... | col"
```

Of course, *tbl*(I) and *eqn*(I)/*neqn*(I), if needed, must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing {11.4} is used with *nroff*, the -c option must be specified to *mm*(I), or the *nroff* output postprocessed by *col*(I). In the latter case, the -T37 terminal type must be specified to *nroff*, the -h option must *not* be specified, and the output of *col*(I) must be processed by the appropriate terminal filter (e.g., *gsi*(I)); *mm*(I) with the -c option handles all this automatically.

2.4 Parameters that Can Be Set from the Command Line

Number registers are commonly used within PWB/MM to hold parameter values that control various aspects of output style. Many of these can be changed within the text files via .nr requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should *not* be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register *P*—see below) *must* be set on the command line (or before the PWB/MM macro definitions are processed) and their meanings are:

-rA1 has the effect of invoking the .AF macro without an argument {6.7.2}.

-rB*n* defines the macros for the cover sheet and the table of contents. If *n* is 1, table-of-contents processing is enabled. If *n* is 2, then cover-sheet processing will occur. If *n* is 3, both will occur. That is, *B* having a value greater than 0 *defines* the .TC {10.1} and/or .CS {10.2} macros. Note that to have any effect, these macros must also be *invoked*.

-rC*n* *n* sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
n = 1 for OFFICIAL FILE COPY.
n = 2 for DATE FILE COPY.
n = 3 for DRAFT.

-rD1 sets *debug mode*. This flag requests the formatter to attempt to continue processing even if PWB/MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 11.3}.

-rL*k* sets the length of the physical page to *k* lines.[1] The default value is 66 lines per page. This parameter is used for obtaining 8 lines-per-inch output on 12-pitch terminals, or when directing output to a Versatec printer.

-rN*n* specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header {9.2}. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, "section-page" numbering {4.5} occurs.

n	*Page 1*	*Pages 2 ff.*
0	header	header
1	header replaces footer	header
2	no header	header
3	"section-page" as *footer*	

The contents of the prevailing header and footer do *not* depend on of the value of the number register *N*; *N* only controls whether and where the header (and, for *N*= 3, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of *N* is irrelevant.

-rO*k* offsets output *k* spaces to the right.[1] It is helpful for adjusting output positioning on some terminals. NOTE: The register name is the capital letter "O", *not* the digit zero (0).

-rP*n* specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.

---

1. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

-rS*n* sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point leading (vertical spacing), giving 6 lines per inch {11.8}. This parameter applies to *troff* only.

-rT*n* provides register settings for certain devices. If *n* is 1, then the line length and page offset are set for output directed to a DASI300 or DASI450 in 12-pitch, 6 lines/inch mode, i.e., they are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to *nroff* only.

-rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This parameter applies to *nroff* only.

-rW*k* page width (i.e., line length and title length) is set to *k*.[2] This can be used to change the page width from the default value of 65 characters (6.5 inches).

2.5 Omission of -mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac.m
remainder of text

In this case, one must *not* use the **-mm** flag (nor the *mm*(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized *before* the .so request, because their values are meaningful only if set before the macro definitions are processed. When using this method, it is best to "lock" into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.nr B 1
.so /usr/lib/tmac.m
.H 1 "INTRODUCTION"
   .
   .
   .
```

specifies, for *nroff*, a line length of 80, a page offset of 10, "section-page" numbering, and table of contents processing.

3. FORMATTING CONCEPTS

3.1 Basic Terms

The normal action of the formatters is to *fill* output lines from one or more input lines. The output lines may be *justified* so that both the left and right margins are aligned. As the lines are being filled words are hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (see .SA {11.2}, *Hy* {3.4}, and the formatter .nf and .fi requests [9]). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the PWB/MM macros cause a break.

While formatter requests can be used with PWB/MM, one must fully understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

2. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

— it is much easier to control (and change at any later point in time) the overall style of the document.

— complicated facilities (such as footnotes or tables of contents) can be obtained with ease.

— the user is insulated from the peculiarities of the formatter language.

A good rule is to use formatter requests only when absolutely necessary [3.9].

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full *sentence must* begin on a new line.

3.2 Arguments and Double Quotes

For any macro call, a *null argument* is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Note that *omitting* an argument is *not* the same as supplying a *null argument* (for example, see the .MT macro in [6.6]). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces *must* be enclosed in double quotes (").[3] Otherwise, it will be treated as several separate arguments.

Double quotes (") are *not* permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (``) and/or two acute accents (´´) instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times; for example, headings are first printed in the text and may be (re)printed in the table of contents.

3.3 Unpaddable Spaces

When output lines are *justified* to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this.

First, one may type a backslash followed by a space (``\ ''). This pair of characters directly generates an *unpaddable space*. Second, one may sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily ``recovered'' by inserting:

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the ``.tr ~'', but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is *not* recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

The formatters (and, therefore, PWB/MM) will automatically hyphenate words, if need be. However, the user may specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator, or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the two-character sequence ``\%'') appears at the beginning or end of a word, the word is *not* hyphenated. Alternatively, it can be used to indicate legal hyphenation point(s) inside a word. In any case, *all* occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator:

.HC [hyphenation-indicator]

---

3. A double quote (") is a *single* character that must not be confused with two apostrophes or acute accents (´´), or with two grave accents (``).

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

.HC ^

Note that any word containing hyphens or dashes—also known as *em* dashes—will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, *even if the formatter hyphenation function is turned off.*

Hyphenation can be turned off in the body of the text by specifying:

.nr Hy 0

once at the beginning of the document. For hyphenation control within footnote text and across pages, see {8.3}.

The user may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word "printout," one may specify:

.hw print-out

3.5 Tabs

The macros .MT {6.6}, .TC {10.1}, and .CS {10.2} use the formatter .ta request to set tab stops, and then restore the *default* values[4] of tab settings. Thus, setting tabs to other than the default values is the user's responsibility.

Note that a tab character is always interpreted with respect to its position on the *input line*, rather than its position on the output line. In general, tab characters should appear only on lines processed in "no-fill" mode {3.1}.

Also note that *tbl*(1) {7.3} changes tab stops, but does *not* restore the default tab settings.

3.6 Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers {9}, headings {4}, and list marks {5}. Users who include BEL characters in their input text (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (•) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". For compatibility with *troff*, a bullet string is provided by PWB/MM. Rather than overstriking, use the sequence:

\•(BU

wherever a bullet is desired. Note that the bullet list (.BL) macros {5.3.3.2} use this string to automatically generate the bullets for the list items.

3.8 Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while *nroff* does not. Those who intend to use *nroff* only may use the minus sign ("-") for all three.

Those who wish mainly to use *troff* should follow the escape conventions of {9}.

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

Dash — Type " -- " for each text dash. These can be left alone for *nroff*, and later globally translated for *troff* to "\(em", namely an em dash (—). Note that the dash list (.DL) macros {5.3.3.3} automatically generate the em dashes for the list items.

4. Every eight characters in *nroff*; every ½ inch in *troff*.

Hyphen Type "-" and use as is for both formatters. *Nroff* will print it as is, and *troff* will print a true hyphen.

Minus Type "\-" for a true minus sign, regardless of formatter. *Nroff* will effectively ignore the "\", while *troff* will print a true minus sign.

3.9 Use of Formatter Requests

Most formatter requests [9] should *not* be used with PWB/MM because PWB/MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests [3.1]. However, some formatter requests *are* useful with PWB/MM, namely:

.af	.br	.ce	.de	.ds	.fi	.hw	.ls	.nf	.nr
.nx	.rm	.rr	.rs	.so	.sp	.ta	.ti	.tl	.tr

The .fp, .lg, and .ss requests are also sometimes useful for *troff.* Use of other requests without fully understanding their implications very often leads to disaster.

4. PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in [5].

4.1 Paragraphs

```
.P [type]
one or more lines of text.
```

This macro is used to begin two kinds of paragraphs. In a *left-justified* paragraph, the first line begins at the left margin, while in an *indented* paragraph, it is indented five spaces (see below).

A document possesses a *default paragraph style* obtained by specifying ".P" before each paragraph that does *not* follow a heading [4.2]. The default style is controlled by the register *Pt.* The initial value of *Pt* is 2, which provides indented paragraphs *except* after headings, lists, and displays, in which case they are left-justified. All paragraphs can be forced to be left-justified by inserting the following at the beginning of the document:

```
.nr Pt 0
```

All paragraphs can be forced to be indented by inserting:

```
.nr Pt 1
```

at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

☛ *Values that specify indentation must be* unscaled *and are treated as "character positions," i.e., as a number of* ens. *In* troff, *an* en *is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In* nroff, *an* en *is equal to the width of a character.*

Regardless of the value of *Pt*, an *individual* paragraph can be forced to be left-justified or indented. ".P 0" always forces left justification; ".P 1" always causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a *list*, the indent (if any) of the paragraph is added to the current list indent [5].

4.2 Numbered Headings

```
.H level [heading-text]
zero or more lines of text
```

The .H macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the most major or highest; level 7 the lowest.

☛ *There is no need for a .P macro after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro. In fact, if a .P follows a .H, the user loses much of the flexibility provided by the .H mechanism {4.2.2.2}.*

4.2.1 Normal Appearance. The normal appearance of headings is as shown in this document. The effect of .H varies according to the *level* argument. First-level headings are *preceded* by two blank lines (one vertical space); all others are *preceded* by one blank line (½ a vertical space).

.H 1 heading-text — gives an underlined (bold) heading *followed* by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

.H 2 heading-text — yields an underlined (bold) heading followed by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

.H *n* heading-text — for $3 \leqslant n \leqslant 7$, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are *run-in* headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

Here are the first few .H calls of {4}:

```
.H 1 "PARAGRAPHS AND HEADINGS"
.H 2 "Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance."
.H 3 "Altering Appearance of Headings."
.H 4 "Pre-Spacing and Page Ejection."
.H 4 "Spacing After Headings."
.H 4 "Centered Headings."
.H 4 "Bold, Italic, and Underlined Headings."
.H 5 "Control by Level."
```

4.2.2 Altering Appearance of Headings. Users satisfied with the default appearance of headings may skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection. A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (½ a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting *Ej* to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to *Ej*.

4.2.2.2 Spacing After Headings. Three registers control the appearance of text immediately following a .H call. They are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (½ a vertical space) is inserted after the heading. Defaults for *Hb* and *Hs* are 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the

heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any *stand-alone* heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register *Hi*. If *Hi* is 0, text is left-justified. If *Hi* is 1 (the *default* value), the text is indented according to the paragraph type as specified by the register *Pt* {4.1}. Finally, if *Hi* is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly. Note that this feature is defeated if a .P macro follows the .H or .HU macro {4.2}.

For example, to cause a blank line (½ a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

4.2.2.3 Centered Headings. The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc*, *and* if it is also stand-alone {4.2.2.2}. *Hc* is 0 initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings.

4.2.2.4.1 Control by Level. Any heading that is underlined by *nroff* is made bold or italic by *troff*. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

Formatter	*HF Code*			*Default*
	1	*2*	*3*	*HF*
nroff	no underline	underline	underline	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, all levels are underlined in *nroff*; in *troff*, levels 1 and 2 are bold, levels 3 through 7 are italic. The user may reset HF as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following would result in five underlined (bold) levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 Nroff Underlining Style. *Nroff* can underline in two ways. The normal style (.ul request) is to underline only letters and digits. The continuous style (.cu request) underlines all characters, including spaces. By default, PWB/MM attempts to use the continuous style on any heading that is to be underlined, is *not* run-in, and is short enough to fit on a single line. If a heading is to be underlined, but is either run-in or is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the -rU1 flag when invoking *nroff* {2.4}.

4.2.2.5 Marking Styles—Numerals and Concatenation.

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The .HM macro (heading mark style) allows this choice to be overridden, thus providing "outline" and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

Value	*Interpretation*
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used "outline" style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

4.3 Unnumbered Headings

```
.HU heading-text
```

.HU is a special case of .H; it is handled in the same way as .H, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

```
.HU heading-text
```

and

```
.H 2 heading-text
```

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up Appendices and other sections that may not fit well into the numbering scheme of the main body of a document {13.2.1}.

4.4 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following three things:

- specifying in the register *Cl* what level headings are to be saved;
- invoking the .TC macro {10.1} at the end of the document;
- and specifying -rB*n* {2.4} on the command line.

Any heading whose level is less than or equal to the value of the register *Cl* (contents level) is saved and later displayed in the table of contents. The default value for *Cl* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" diagnostic {Appendix E} will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and the Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the form "section-page," where *section* is the number of the current first-level heading. This page numbering style can be achieved by specifying the flag -rN3 on the command line {9.9}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each section

begins on a new page. In this style, the page number is printed at the *bottom* of the page, so that the correct section number is printed.

4.6 User Exit Macros •

☛ *This section is intended only for users who are accustomed to writing formatter macros.*

.HX dlevel rlevel heading-text
.HZ dlevel rlevel heading-text

The .HX and .HZ macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. PWB/MM does not define .HX and .HZ; they are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. All the default actions occur if these macros are not defined. If the .HX or .HZ (or both) are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time .H calls .HX, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank line(s) (vertical space) to precede the heading {4.2.2.1}, and accumulated the "heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

string }0 — If *rlevel* is non-zero, this string contains the "heading mark." Two unpaddable spaces (to separate the *mark* from the *heading*) have been appended to this string. If *rlevel* is 0, this string is null.

register ;0 — This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (½ a vertical space) is to follow the heading.

string }2 — If register ;0 is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.

register ;3 — This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of *}0*, *}2*, and *;3* within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format *n.* to *n*.0:
.if \\$1=1 .ds }0 \\n(H1.0\□\□ (□ stands for a space)

Separate run-in heading from the text with a period and two unpaddable spaces:
.if \\n(;0=0 .ds }2 .\□\□

Assure that at least 15 lines are left on the page before printing a first-level heading:
.if \\$1=1 .nr ;3 15-\\n(;0

Add 3 additional blank lines before each first-level heading:
.if \\$1=1 .sp 3

If temporary string or macro names are used within .HX, care must be taken in the choice of their names {13.1}.

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections may correspond to chapters of a book, and the user may want to reset counters for footnotes, figures, tables, etc. Another use might be to change a page header or footer. For example:

```
.de HZ
.if \\$1=1 \{.nr :p 0 \" footnotes
.	nr Fg 0 \" figures
.	nr Tb 0 \" tables
.	nr Ec 0 \" equations
.	PF "''Section \\$3''"\}
..
```

4.7 Hints for Large Documents

A large document is often organized for convenience into one file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register *H1* to 1 less than the number of the section just *before* the corresponding ".H 1" call. For example, at the beginning of section 5, insert:

```
.nr H1 4
```

☛ *This is a dangerous practice: it defeats the automatic (re)numbering of sections when sections are added or deleted. Remove such lines as soon as possible.*

5. LISTS

This section describes many different kinds of lists: automatically-numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, e.g., with terms or phrases to be defined.

5.1 Basic Approach

In order to avoid repetitive typing of arguments to describe the appearance of items in a list, PWB/MM provides a convenient way to specify lists. All lists are composed of the following parts:

- A *list-initialization* macro that controls the appearance of the list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more *List Item* (.LI) macros, each followed by the actual text of the corresponding list item.
- The *List End* (.LE) macro that terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of that list. In addition, by building on the existing structure, users may create their own customized sets of list macros with relatively little effort {5.4, Appendix A, Appendix B}.

5.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls {5.3.3} contained therein are examples of the *list-initialization* macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a "plus" as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```

Output:

A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

   1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

      — This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

      + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

   2. This is numbered item 2.

B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.3 Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1 List Item.

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The .LI macro is used with all lists. It normally causes the output of a single blank line (½ a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the *current mark*, which is specified by the most recent list-initialization macro. If a single argument is given to .LI, that argument is output *instead of* the current mark. If two arguments are given, the first argument becomes a *prefix* to the current mark, thus allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a "plus."
.LI + xxx
But this uses "plus" as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.

\+ This replaces the bullet with a "plus."

\+ • But this uses "plus" as prefix to the bullet.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

If the *current mark* (in the *current list*) is a null string, and the first argument of .LI is omitted or null, the resulting effect is that of a *hanging indent*, i.e., the first line of the following text is "outdented," starting at the same place where the *mark* would have started {5.3.3.6}.

5.3.2 List End.

.LE [1]

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the .LE outputs a blank line (½ a vertical space). This option should generally be used only when the .LE is followed by running text, but not when followed by a macro that produces blank lines of its own, such as .P, .H, or .LI.

.H and .HU automatically clear all list information, so one may legally omit the .LE(s) that would normally occur just before either of these macros. Such a practice is *not* recommended, however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3 List Initialization Macros. The following are the various list-initialization macros. They are actually implemented as calls to the more basic .LB macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists.

.AL [type] [text-indent] [1]

The .AL macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text is indented *Li* (initially 5)[5] spaces from the indent in force when the .AL is called, thus leaving room for two digits, a period, and two spaces before the text.

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i {4.2.2.5}.[6] If *type* is omitted or null, then "1" is assumed. If *text-indent* is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

---

5. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

6. Note that the "0001" format is *not* permitted.

If the third argument is given, a blank line (½ a vertical space) will *not* separate the items in the list. A blank line (½ a vertical space) will occur before the first item, however.

5.3.3.2 Bullet List.

.BL [text-indent] [1]

.BL begins a bullet list, in which each item is marked by a bullet (•) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi* {4.1}.[7]

If a second argument is specified, no blank lines will separate the items in the list.

5.3.3.3 Dash List.

.DL [text-indent] [1]

.DL is identical to .BL, except that a dash is used instead of a bullet.

5.3.3.4 Marked List.

.ML mark [text-indent] [1]

.ML is much like .BL and .DL, but expects the user to specify an arbitrary mark, which may consist of more than a single character. Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*. If the third argument is specified, no blank lines will separate the items in the list.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

5.3.3.5 Reference List.

.RL [text-indent] [1]

A .RL call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). *Text-indent* may be supplied, as for .AL. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list. The list of references {14} was produced using the .RL macro.

5.3.3.6 Variable-Item List.

.VL text-indent [mark-indent] [1]

When a list begins with a .VL, there is effectively no *current mark;* it is expected that each .LI will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark*, and it defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of .VL usage:

---

7. So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

```
.tr ~
.VL 20 2
.LI mark~1
Here is a description of mark 1;
"mark 1" of the .LI line contains a tilde translated to an unpaddable space in order
to avoid extra spaces between
"mark" and "1" (3.3).
.LI second~mark
This is the second mark, also using a tilde translated to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1 — Here is a description of mark 1; "mark 1" of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between "mark" and "1" (3.3).

second mark — This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent: This item shows the effect of a long mark; one space separates the mark from the text.

This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a *hanging indent* would have been produced. A *hanging indent* is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is
indented 10 spaces.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* (3.3).

5.4 List-Begin Macro and Customized Lists •

.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

The list-initialization macros described above suffice for almost all cases. However, if necessary, one may obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros (Appendix A). Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bullet and dash lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and

ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).[8] Within the mark area, the mark is *left-justified* if *pad* is 0. If *pad* is greater than 0, say *n*, then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively *right-justified pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.3.3.1}. That is:

Type	*Mark*	*Result*
0	omitted	hanging indent
0	*string*	*string* is the mark
>0	omitted	arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the items. The following table shows the output appearance for each value of *type:*

Type	*Appearance*
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where *x* is the generated number or letter.

☞ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each .LI macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the .LI macro issues a .ne request for two lines just before printing the mark.

LB-space, the number of blank lines (½ a vertical space) to be output by .LB itself, defaults to 0 if omitted.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a ".LE 1" to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use ".LE 1" at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use ".LE" to end the list, a list without *any* blank lines will result.

Appendix A shows the definitions of the list-initialization macros {5.3.3} in terms of the .LB macro. Appendix B illustrates how the user can build upon those macros to obtain other kinds of lists.

6. MEMORANDUM AND RELEASED PAPER STYLES

One use of PWB/MM is for the preparation of memoranda and released papers, which have special requirements for the first page and for the cover sheet. The information needed for the memorandum or released paper (title, author, date, case numbers, etc.) is entered in the same way for *both* styles; an argument to one macro indicates which style is being used. The following sections describe the macros used to provide this data. The required order is shown in {6.9}.

8. The *mark-indent* argument is typically 0.

If neither the memorandum nor released-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header {9} followed by the body of the document.

6.1 Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

The arguments to the .TL macro are the charging case number(s) and filing case number(s).[9] The title of the memorandum or paper follows the .TL macro and is processed in fill mode {3.1}. Multiple charging case numbers are entered as "sub-arguments" by separating each from the previous with a comma and a space, and enclosing the *entire* argument within double quotes. Multiple filing case numbers are entered similarly. For example:

```
.TL "12345, 67890" 987654321
On the construction of a table
of all even prime numbers
```

The .br request may be used to break the title into several lines.

On output, the title appears after the word "subject" in the memorandum style. In the released-paper style, the title is centered and underlined (bold).

6.2 Author(s)

```
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
```

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given (a separate .AU macro is required for each author). For example:

```
.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9, if present, will follow this "normal" author information, each on a separate line. Certain organizations have their own numbering schemes for memoranda, engineer's notes, etc. These numbers are printed after the author's name. This can be done by providing more than six arguments to the .AU macro, e.g.:

```
.AU "S. P. Lename" SPL IH 9988 7766 5H-444 3322.11AB
```

The name, initials, location, and department are also used in the Signature Block {6.11.1}. The author information in the "from" portion, as well as the names and initials in the Signature Block will appear in the same order as the .AU macros.

The names of the authors in the released-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For the case of authors from different locations, see {6.8}.

6.3 TM Number(s)

```
.TM [number] ...
```

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers may be specified. Example:

```
.TM 7654321 77777777
```

This macro call is ignored in the released-paper and external-letter styles {6.6}.

9. The "charging case" is the case number to which time was charged for the development of the project described in the memorandum. The "filing case" is a number under which the memorandum is to be filed.

6.4 Abstract

```
.AS [arg] [indent]
text of the abstract
.AE
```

In both the memorandum and released-paper styles, the text of the abstract follows the author information and is preceded by the centered and underlined (italic) word "ABSTRACT."

The .AS (abstract start) and .AE (abstract end) macros bracket the (optional) abstract. The first argument to .AS controls the printing of the abstract. If it is 0 or null, the abstract is printed on the first page of the document, immediately following the author information, and is also saved for the cover sheet. If the first argument is 1, the abstract is saved and printed only on the cover sheet. The margins of the abstract are indented on the left and right by five spaces. The amount of indentation can be changed by specifying the desired indentation as the second argument.[10]

Note that headings {4.2, 4.3}, displays {7}, and footnotes {8} are *not* (as yet) permitted within an abstract.

6.5 Other Keywords

```
.OK [keyword] ...
```

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6 Memorandum Types

```
.MT [type] [1]
```

The .MT macro controls the format of the top part of the first page of a memorandum or of a released paper, as well as the format of the cover sheets. Legal codes for *type* and the corresponding values are:

Code	*Value*
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-Paper style
.MT 5	External-Letter style
.MT "*string*"	*string*

If *type* indicates a memorandum style, then *value* will be printed after the last line of author information or after the last line of the abstract, if one appears on the first page. If *type* is longer than one character, then it, itself, will be printed. For example:

```
.MT "Technical Note #5"
```

A simple letter is produced by calling .MT with a null (but *not* omitted!) or zero argument.

The second argument to .MT is used only if the first argument is 4 (i.e., for the released-paper style) as explained in {6.8}.

In the external-letter style (.MT 5), only the date is printed in the upper right corner of the first page. It is expected that preprinted stationery will be used, providing the author's company logotype and address.

---

10. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

6.7 Date and Format Changes

6.7.1 Changing the Date. By default, the current date appears in the "date" part of a memorandum. This can be overridden by using:

.ND new-date

The .ND macro alters the value of the string *DT*, which is initially set to the current date.

6.7.2 Alternate First-Page Format. One can specify that the words "subject," "date," and "from" (in the memorandum style) be omitted and that an alternate company name be used:

.AF [company-name]

If an argument is given, it replaces "Bell Laboratories", without affecting the other headings. If the argument is *null*, "Bell Laboratories" is suppressed; in this case, extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp. .AF with *no* argument suppresses "Bell Laboratories" and the "Subject/Date/From" *headings*, thus allowing output on preprinted stationery.

The only .AF option appropriate for *troff* is to specify an argument to replace "Bell Laboratories" with another name.

6.8 Released-Paper Style

The released-paper style is obtained by specifying:

.MT 4 [1]

This results in a centered, underlined (bold) title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF {6.7.2} specifies a different company). If the optional second argument to .MT is given, then the name of each author is followed by the respective company name and location. The abstract, if present, follows the author information.

Information necessary for the memorandum style but not for the released-paper style is ignored.

If the released-paper style is utilized, most BTL location codes[11] are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, *following* the .MT macro, the user may re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF *before* each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.AU "F. Swatter"
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

6.9 Order of Invocation of "Beginning" Macros

The macros described in {6.1-6.7}, *if present*, must be given in the following order:

11. The complete list is: AK, CP, CH, CB, DR, HO, IN, IH, MV, MH, PY, RR, RD, WV, and WH.

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.OK [keyword] ...
.MT [type] [1]
```

The only *required* macros for a memorandum or a released paper are .TL, .AU, and .MT; all the others (and their associated input lines) may be omitted if the features they provide are not needed. Once .MT has been invoked, *none* of the above macros can be re-invoked because they are removed from the table of defined macros to save space.

6.10 Example

The input text for this manual begins as follows:

```
.TL
P\s-3WB/MM\s0\(emProgrammer's Workbench Memorandum Macros
.AU "D. W. Smith" DWS PY ...
.AU "J. R. Mashey" JRM  MH ...
.MT 4 1
```

6.11 Macros for the End of a Memorandum

At the end of a memorandum (but not of a released paper), the signatures of the authors and a list of notations[12] can be requested. The following macros and their input are ignored if the released-paper style is selected.

6.11.1 Signature Block.

```
.SG [arg] [1]
```

.SG prints the author name(s) after the last line of text, aligned with the "Date/From" block. Three blank lines are left above each name for the actual signature. If no argument is given, the line of reference data[13] will *not* appear following the last line.

A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null argument to print reference data with neither the typist's initials nor the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first author, rather than on the line that has the name of the last author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

12. See [2], pp. 1.12–16

13. The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens. See [2], page 1.11

6.11.2 "Copy to" and Other Notations.

```
.NS [arg]
zero or more lines of the notation
.NE
```

After the signature and reference data, many types of notations may follow, such as a list of attachments or "copy to" lists. The various notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

Code	*Notations*
.NS ""	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to
.NS "*string*"	Copy (*string*) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to." For example:

```
.NS "with att. 1 only"
```

will generate "Copy (with att. 1 only) to" as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as:

Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz

6.12 Forcing a One-Page Letter

At times, one would like just a bit more space on the page, forcing the signature or items within not tions onto the bottom of the page, so that the letter or memo is just one page in length. This can I accomplished by increasing the page length through the -rL*n* option, e.g. -rL90. This has the effect

making the formatter believe that the page is 90 lines long and therefore giving it more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

7. DISPLAYS

Displays are blocks of text that are to be kept together—not split across pages. PWB/MM provides two styles of displays:[14] a *static* (.DS) style and a *floating* (.DF) style. In the *static* style, the display appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the *floating* style, the display "floats" through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that *follows* a floating display may *precede* it in the output text. A queue of floating displays is maintained so that their relative order is not disturbed.

By default, a display is processed in no-fill mode and is *not* indented from the existing margin. The user can specify indentation or centering, as well as fill-mode processing.

Displays and footnotes {8} may *never* be nested, in any combination whatsoever. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) can occur within displays or footnotes.

7.1 Static Displays

```
.DS [format] [fill]
one or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will *not* indent them from the prevailing indentation. The *format* argument to .DS is an integer with the following meanings:

Code	*Meaning*
""	no indent
0	no indent
1	indent by standard amount
2	center each line

The *fill* argument is also an integer and can have the following meanings:

Code	*Meaning*
""	no-fill mode
0	no-fill mode
1	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register *Si*, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same, these two registers are independent of one another.

By default, a blank line (½ a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

7.2 Floating Displays

```
.DF [format] [fill]
one or more lines of text
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as for .DS {7.1}, except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change

14. Displays are processed in an environment that is different from that of the body of the text (see the .ev request in [9]).

between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (½ a vertical space) *always* occurs both before and after a floating display.

7.3 Tables

```
.DS
.TS
one or more lines of text to be processed by tbl(1)
.TE
.DE
```

The .TS (table start) and .TE (table end) macros make possible the use of the *tbl*(1) processor [11]. They are used *only* to delimit the text to be examined by *tbl*(1). Thus, the display function and the *tbl*(1) delimiting function are independent of one another, in order to permit one to keep together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines.

If a particular document does not need this flexibility, it is possible to define .TS and .TE so that they act like .DS and .DE, respectively, and are also recognized by *tbl*(1):

```
.de TS
.Ds "\\$1" "\\$2"
..
.de TE
.DE
..
```

If floating tables are desired, substitute .DF for .DS in the above.

7.4 Equations

```
.DS
.EQ
equation(s)
.EN
.DE
```

The equation setters *eqn*(1) and *neqn*(1) [6,7] expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that *tbl*(1) uses .TS and .TE; .EQ and .EN must occur either inside .DS-.DE pairs or else be defined by the user as shown above for the .TS and .TE macros.

☞ *There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify* eqn/neqn *"defines," .DS and .DE must* not *be used; otherwise extra blank lines will appear in the output.*

7.5 Figure, Table, and Equation Captions

```
.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
```

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) macros are normally used insid .DS-.DE pairs to automatically number and title figures, tables, and equations. They use registers *Fg*, *Tb*, and *Ec*, respectively.[15] As an example, the call:

```
.FG "This is an illustration"
```

yields:

Figure 1. This is an illustration

.TB replaces "**Figure**" by "**TABLE**"; .EC replaces "**Figure**" by "**Equation**". Output is centered if can fit on a single line; otherwise, all lines but the first are indented to line up with the first character the title. The format of the numbers may be changed using the .af request of the formatter.

15. The user may wish to reset these registers after each first-level heading [4.6].

The *override* string is used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. For example, to produce figures numbered within sections, supply \n(H1 for *override* on each .FG call, and reset *Fg* at the beginning of each section, as shown in {4.6}.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure and equation captions usually occur after the corresponding figures and equations.

7.6 Blocks of Filled Text

One can obtain blocks of filled text through the use of .DS or .DF. However, to have the block of filled text *centered* within the current line length, the *tbl*(I) program may be used:

```
  :
.DS 0 1
.TS
center;
lw40 .
T{
  :
T}
.TE
.DE
  :
```

The ".DS 0 1" begins a non-indented, filled display. The *tbl*(I) parameters set up a centered table with a column width of 40 ens. The "T{ ... T}" sequence allows filled text to be input as data within a table.

8. FOOTNOTES

There are two macros that delimit the text of footnotes,[16] a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "*F" immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

8.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The .FS (footnote start) marks the beginning of the text of the footnote, and the .FE marks its end. The *label* on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted *must* be followed by *label*, rather than by "*F". The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are *not* permitted between the .FS and .FE macros. Examples:

16. Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request in [9]).

1. Automatically-numbered footnote:

   This is the line containing the word*F
   .FS
   This is the text of the footnote.
   .FE
   to be footnoted.

2. Labelled footnote:

   This is a labeled*
   .FS *
   The footnote is labeled with an asterisk.
   .FE
   footnote.

The text of the footnote (enclosed within the .FS-.FE pair) should *immediately* follow the word to be footnoted in the input text, so that "*F" or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append a unpaddable space {3.3} to "*F" or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix C illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format of Footnote Text •

.FD [arg] [1]

Within the footnote text, the user can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The .FD macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests in [9].

0	.nh	.ad	text indent	label left justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to .FD is out of range, the effect is as if .FD 0 were specified. If the first argument is omitted or null, the effect is equivalent to .FD 10 in *nroff* and to .FD 0 in *troff*; these are also the respective initial defaults.

If a second argument is specified, then whenever a first-level heading is encountered, automatically numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line:

.FD "" 1

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (over which the user has control by specifying an *even* argument to .FD), hyphenation across pages is inhibited by PWB/MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In *troff,* footnotes are set in type that is two points smaller than the point size used in the body of the text.

8.4 Spacing between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register *Fs* to the desired value. For example:

.nr Fs 2

will cause two blank lines (a six-point vertical space) to occur between footnotes.

9. PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the *page header.* Text printed at the bottom of each page is called the *page footer.* There can be up to three lines of text associated with the header: every page, even page only, and odd page only. Thus the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term *header (not* qualified by *even* or *odd)* to mean the line of the page header that occurs on every page, and similarly for the term *footer.*

9.1 Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a released paper, the page header on the first page is automatically suppressed *provided* a break does *not* occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do *not* cause a break and are permitted before the .MT macro call.

9.2 Page Header

.PH [arg]

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

"'left-part'center-part'right-part'"

If it is inconvenient to use the apostrophe (') as the delimiter (i.e., because it occurs within one of the parts), it may be replaced *uniformly* by *any* other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. See the top of this page for an example of this default header.

If *debug mode* is set using the flag -rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS [10] Release and Level of PWB/MM (thus identifying the current version {11.3}), followed by the current line number within the current input file.

9.3 Even-Page Header

.EH [arg]

The .EH macro supplies a line to be printed at the top of each even-numbered page, immediately *following* the header. The initial value is a blank line.

9.4 Odd-Page Header

.OH [arg]

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rC*n* flag is specified on the command line {2.4}, the type of copy *follows* the footer on a separate line. In particular, if -rC3 (DRAFT) is specified, then, in addition, the footer is initialized to contain the date {6.7.1}, instead of being a blank line.

9.6 Even-Page Footer

.EF [arg]

The .EF macro supplies a line to be printed at the bottom of each even-numbered page, immediately *preceding* the footer. The initial value is a blank line.

9.7 Odd-Page Footer

.OF [arg]

This macro is the same as .EF, except that it applies to odd-numbered pages.

9.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies .PF and/or .OF before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) *replaces* the footer *on the first page only* if the -rN1 flag is specified on the command line {2.4}.

9.9 Default Header and Footer with "Section-Page" Numbering

Pages can be numbered sequentially within sections {4.5}. To obtain this numbering style, specify -rN3 on the command line. In this case, the default *footer* is a centered "section-page" number, e.g. 3-5, and the default page header is blank.

9.10 Use of Strings and Registers in Header and Footer Macros •

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed *when the respective header or footer is printed*, the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a .tl request during header or footer processing.

For example, the page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH "'''Page \\\\nP'"
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, one specifies (see {4.2.2.5} for meaning of *H1*):

```
.PF "'''- \\\\n(H1-\\\\nP -'"
```

As another example, suppose that the user arranges for the string *a]* to contain the current section heading which is to be printed at the bottom of each page. The .PF macro call would then be:

```
.PF "''\\\\*(a]''"
```

If only one or two backslashes were used, the footer would print a constant value for *a]*, namely, its value when the .PF appeared in the input text.

9.11 Header and Footer Example •

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH ""
.PF ""
.EH "'\\\\nP''Revision 3'"
.OH "'Revision 3''\\\\nP'"
```

9.12 Generalized Top-of-Page Processing •

☛ *This section is intended only for users accustomed to writing formatter macros.*

During header processing, PWB/MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request in [9]) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with "no-space" mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
.sp
.tl \\*(}t
.if e 'tl \\*(}e
.if o 'tl \\*(}o
.sp
..
```

The string }*t* contains the header, the string }*e* contains the even-page header, and the string }*o* contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing {11.5}. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.13 Generalized Bottom-of-Page Processing

The facility to permit user-defined processing for the bottom of each page is *not* currently available.

10. TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively. The appropriate -rB*n* option {2.4} must *also* be specified on the command line. These macros should normally appear only once at the *end* of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1 Table of Contents

.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *Cl* register (4.4). Note that -rB1 or -rB3 (2.4) must also be specified to the formatter on the command line. The arguments to .TC control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (½ a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *Cl* register (4.4).

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots ("leaders") separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are "ragged right").

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked with at most four arguments, then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed. By defining .TX and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: ____________________________________

CONTENTS
⋮

10.2 Cover Sheet

.CS [pages] [other] [total] [figs] [tbls] [refs]

The .CS macro generates a cover sheet in either the TM or released-paper style.[17] All of the other information for the cover sheet is obtained from the data given before the .MT macro call (6.9). If the released-paper style is used, all arguments to .CS are ignored. If a memorandum style is used, the .CS macro generates the "Cover Sheet for Technical Memorandum." The arguments provide the data that appears in the lower left corner of the TM cover sheet [2]: the number of pages of text, the number of other pages, the total number of pages, the number of figures, the number of tables, and the number of references.

11. MISCELLANEOUS FEATURES

11.1 Bold, Italic, and Roman

.B [bold-arg] [previous-font-arg]
.I [italic-arg] [previous-font-arg]
.R

When called without arguments, .B (or .I) changes the font to bold (or italic) in *troff*, and initiates underlining in *nroff*.[18] This condition continues until the occurrence of a .R, when the regular roman font is restored. Thus,

.I
here is some text.
.R

yields:

here is some text.

If .B or .I is called with one argument, that argument is printed in the appropriate font (underlined in *nroff*). Then the *previous* font is restored (underlining is turned off in *nroff*). If two arguments are given to a .B or .I, the second argument is then concatenated to the first with no intervening space, but is printed in the previous font (not underlined in *nroff*). For example:

.I italic
text
.I right -justified

produces:

italic text *right*-justified

One can use both bold and italic fonts if one intends to use *troff*, but the *nroff* version of the output does not distinguish between bold and italic. It is probably a good idea to use .I only, unless bold is truly required. Note that font changes in headings are handled separately (4.2.2.4.1).

Anyone using a terminal that cannot underline might wish to insert:

.rm ul
.rm cu

at the beginning of the document to eliminate *all* underlining.

11.2 Justification of Right Margin

.SA [arg]

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. .SA 0 sets both flags to no justification, i.e., it acts like the .na request. .SA 1 is the inverse: it sets both flags to cause justification, just like the .ad request. However, calling

17. But only if -rB2 or -rB3 has been specified on the command line.

18. For ease of explanation, in this section (11.1) *troff* behavior is described first, the convention of (1.2) not withstanding.

.SA *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a .na or .ad, depending on what the *default* is. Initially, both flags are set for no justification in *nroff* and for justification in *troff.*

In general, the request .na can be used to ensure that justification is turned off, but .SA should be used to restore justification, rather than the .ad request. In this way, justification or lack thereof for the remainder of the text is specified by inserting .SA 0 or .SA 1 *once* at the beginning of the document.

11.3 SCCS Release Identification

The string *RE* contains the SCCS [10] Release and Level of the current version of PWB/MM. For example, typing:

This is version *(RE of the macros.

produces:

This is version 12.2 of the macros.

This information is useful in analyzing suspected bugs in PWB/MM. The easiest way to have this number appear in your output is to specify -rD1 {2.4} on the command line, which causes the string *RE* to be output as part of the page header {9.2}.

11.4 Two-Column Output

PWB/MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar "pages" of equal (but smaller) "page" width. Page headers and footers are *not* affected by two-column processing. The .1C macro does *not* "balance" two-column output.

11.5 Column Headings for Two-Column Output •

☞ *This section is intended only for users accustomed to writing formatter macros.*

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page {9}. This is accomplished by redefining the .TP macro {9.12} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP'OVERALL''
.tl ''TITLE''
.sp
.nf
.ta 16C 31R 34 50C 65R
left→center→right→left→center→right          (where → stands for the tab character)
→first column→→→second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

11.6 Vertical Spacing

.SP [lines]

There exist several ways of obtaining vertical spacing, all with different effects.

The .sp request spaces the number of lines specified, *unless* "no space" (.ns) mode is on, in which case the request is ignored. This mode is typically set at the end of a page header in order to eliminate spacing by a .sp or .bp request that just happens to occur at the top of a page. This mode can be turned *off* via the .rs ("restore spacing") request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many PWB/MM macros utilize .SP for spacing. For example, ".LE 1" {5.3.2} immediately followed by ".P" {4.1} produces only a single blank line (½ a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Unscaled fractional amounts are permitted; like .sp, .SP is also inhibited by the .ns request.

11.7 Skipping Pages

.SK [pages]

The .SK macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, .SK skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. .SK *n* skips *n* pages. That is, .SK always positions the text that follows it at the top of a page, while .SK 1 always leaves one page that is blank except for the header and footer.

11.8 Setting Point Size and Vertical Spacing

In *troff*, the default point size (obtained from the register *S* {2.4}) is 10, with a vertical spacing of 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the .S macro:

.S [arg]

If *arg* is null, the *previous* point size is restored. If *arg* is negative, the point size is decremented by the specified amount. If *arg* is *signed* positive, the point size is incremented by the specified amount, and if *arg* is unsigned, it is used as the new point size; if *arg* is greater than 99, the *default* point size (10) is restored. Vertical spacing is always two points greater than the point size.[19]

12. ERRORS AND DEBUGGING

12.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix E.)
- Processing terminates, unless the register *D* {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

☛ *The error message is printed by writing it directly to the user's terminal. If an output filter, such as* gsi(1), 450(1), *or* hp(1) *is being used to post-process* nroff *output, the message may be garbled by being intermixed with text held in that filter's output buffer.*

---

19. Footnotes {8} are printed in a size two points *smaller* than the point size of the body, with an additional vertical spacing of three points between footnotes.

☛ *If either* tbl(I) *or* eqn(I)/neqn(I), *or both are being used, and if the* -olist *option of the formatter causes the last page of the document* not *to be printed, a harmless "broken pipe" message results.*

12.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .FE or .DE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is to search backwards from the termination point looking for the corresponding .DS, .DF, or .FS.

The following command:

grep -n "^\.[EDFT][EFNQS]" files ...

prints all the .DS, .DF, .DE, .FS, .FE, .TS, .TE, .EQ, and .EN macros found in *files* ..., each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

13. EXTENDING AND MODIFYING THE MACROS •

13.1 Naming Conventions

In this section, the following conventions are used to describe legal names:

n: digit
a: lower-case letter
A: upper-case letter
x: any letter or digit (any alphanumeric character)
s: special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request, macro,* and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

13.1.1 Names Used by Formatters.

requests: aa (most common)
an (only one, currently: .c2)

registers: aa (normal)
.x (normal)
.s (only one, currently: .$)
% (page number)

13.1.2 Names Used by PWB/MM.

macros: AA (most common, accessible to user)
A (less common, accessible to user)
)x (internal, constant)
>x (internal, dynamic)

strings: AA (most common, accessible to user)
A (less common, accessible to user)
]x (internal, usually allocated to specific functions throughout)
}x (internal, more dynamic usage)

registers: Aa (most common, accessible to users)
An (common, accessible to user)
A (accessible, set on command line)
:x (mostly internal, rarely accessible, usually dedicated)
;x (internal, dynamic, temporaries)

13.1.3 Names Used by EQN/NEQN and TBL. The equation preprocessors, *eqn*(I) and *neqn*(I), use registers *and* string names of the form *nn.* The table preprocessor, *tbl*(I), uses names of the form:

a- a+ a| nn #a ## #- #^ ^a T& TW

13.1.4 User-Definable Names. After the above, what is left for user extensions? To avoid problems, we suggest using names that consist either of a single lower-case letter, or of a lower-case letter followed by anything other than a lower-case letter. The following is a sample naming convention:

macros:	aA
	Aa
strings:	a
	a) (or a], or a}, etc.)
registers	a
	aA

13.2 Sample Extensions

13.2.1 Appendix Headings. The following gives a way of generating and numbering appendices:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH "'''Appendix \\na - \\\\\\\\nP'"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ".aH "title"" begins a new page (with the page header changed to "Appendix *a - n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish Appendix titles to be centered must, in addition, set the register *Hc* to 1 {4.2.2.3}.

13.2.2 Hanging Indent with Tabs. The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the *mark.* Each argument is to be separated from the previous one by a tab character; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro.[20] The "\t" is translated by the formatter into a tab character. The "\c" is used to concatenate the line of *text* that follows the macro to the line of text built by the macro. The macro definition and an example of its use are as follows:

20. The two-character sequence "\&" is understood by the formatters to be a "zero-width" space, i.e., it causes no output characters to appear.

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
..
   .
   .
   .
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\□c none none no                          (□ stands for a space)
Hyphenation indicator character is set to "c" or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	–	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	–	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to "c" or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

14. CONCLUSION

The following are the qualities that we have tried to emphasize in PWB/MM, in approximate order of importance:

- *Robustness in the face of error*—A user need not be an *nroff/troff* expert to use these macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error, or a message describing the error is produced. We have tried to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce simple documents. Reasonable default values are provided, where at all possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt the output to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. Naming conventions are given so that a user can add new macros or redefine existing ones, if necessary.
- *Device independence*—The most common use of PWB/MM is to print documents on hard-copy typewriter terminals, using the *nroff* formatter. The macros can be used conveniently with both 10- and

12-pitch terminals. In addition, output can be scanned with an appropriate CRT terminal. The macros have been constructed to allow compatibility with *troff*, so that output can be produced both on typewriter-like terminals and on a phototypesetter.

- *Minimization of input*—The design of the macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, he or she need only set a specific parameter *once* at the beginning of a document, rather than add a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text, although the user may obtain a number of output styles by setting a few global flags. For example, the .H macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or, for that matter, within a single document.

Future releases of PWB/MM will provide additional features that are found to be useful. The authors welcome comments, suggestions, and criticisms of the macros and of this manual.

Acknowledgements. We are indebted to T. A. Dolotta for his continuing guidance during the development of PWB/MM. We also thank our many users who have provided much valuable feedback, both about the macros and about this manual. Many of the features of PWB/MM are patterned after similar features in a number of earlier macro packages, and, in particular, after one implemented by M. E. Lesk. Finally, because PWB/MM often approaches the limits of what is possible with the text formatters, during the implementation of PWB/MM we have generated atypical requirements and encountered unusual problems; we thank J. F. Ossanna for his willingness to add new features to the formatters and to invent ways of having the formatters perform unusual but desired actions.

References

[1] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[2] Bell Laboratories, Methods and Systems Department. Office Guide. Unpublished Memorandum, Bell Laboratories, April 1972 (as revised).

[3] Kernighan, B. W. UNIX for Beginners. Bell Laboratories, October 1974.

[4] Kernighan, B. W. A Tutorial Introduction to the UNIX Text Editor. Bell Laboratories, October 1974.

[5] Kernighan, B. W. A TROFF Tutorial. Bell Laboratories, August 1976.

[6] Kernighan, B. W., and Cherry, L. L. Typesetting Mathematics—User's Guide (Second Edition). Bell Laboratories, June 1976.

[7] Scrocca, C. New Graphic Symbols for EQN and NEQN. Bell Laboratories, September 1976.

[8] Smith, D. W., and Piskorik, E. M. Typing Documents with PWB/MM. Bell Laboratories, October 1977.

[9] Ossanna, J. F. NROFF/TROFF User's Manual. Bell Laboratories, October 1976.

[10] Bonanni, L. E., and Glasser, A. L. SCCS/PWB User's Manual. Bell Laboratories, November 1977.

[11] Lesk, M. Tbl—A Program to Format Tables. Bell Laboratories, September 1977.

Appendix A: DEFINITIONS OF LIST MACROS •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

Here are the definitions of the list-initialization macros (5.3.3):[21]

```
.de AL
.if!@\\$1@@ .if!@\\$1@1@ .if!@\\$1@a@ .if!@\\$1@A@ .if!@\\$1@I@ .if!@\\$1@i@ .)D"AL:bad arg:\\$1"
.if \\n(.$<3 \{.ie \w@\\$2@=0 .)L \\n(Lin 0 \\n(Lin-\w@\0\0.@u 1 "\\$1"
.el .LB 0\\$2 0 2 1 "\\$1" \}
.if \\n(.$>2 \{.ie \w@\\$2@=0 .)L \\n(Lin 0 \\n(Lin-\w@\0\0.@u 1 "\\$1" 0 1
.el .LB 0\\$2 0 2 1 "\\$1" 0 1 \}
..
.de BL
.nr ;0 \\n(Pi
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 1 0 \\*(BU
.if \\n(.$>1 .LB \\n(;0 0 1 0 \\*(BU 0 1
.rr ;0
..
.de DL
.nr ;0 \\n(Pi
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 1 0 \(em
.if \\n(.$>1 .LB \\n(;0 0 1 0 \(em 0 1
.rr ;0
..
.de ML
.if !\\n(.$ .)D "ML:missing arg"
.nr ;0 \w@\\$1@u/3u/\\n(.su+1u\" get size in n's
.if !\\n(.$-1 .LB \\n(;0 0 1 0 "\\$1"
.if \\n(.$-1 .if !\\n(.$-2 .LB 0\\$2 0 1 0 "\\$1"
.if \\n(.$-2 .if !\w@\\$2@ .LB \\n(;0 0 1 0 "\\$1" 0 1
.if \\n(.$-2 .if \w@\\$2@ .LB 0\\$2 0 1 0 "\\$1" 0 1
..
.de RL
.nr ;0 6
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 2 4
.if \\n(.$>1 .LB \\n(;0 0 2 4 1 0 1
.rr ;0
..
.de VL
.if !\\n(.$ .)D "VL:missing arg"
.if !\\n(.$-2 .LB 0\\$1 0\\$2 0 0
.if \\n(.$-2 .LB 0\\$1 0\\$2 0 0 \& 0 1
..
```

Any of these can be redefined to produce different behavior: e.g., to provide two spaces between the bullet of a bullet item and its text, redefine .BL as follows before invoking it:[22]

```
.de BL
.LB 3 0 2 0 \\*(BU
..
```

21. On this page, @ represents the BEL character, .)D is an internal PWB/MM macro that prints error messages, and .)L is similar to .LB, except that it expects its arguments to be scaled.

22. With this redefinition, .BL cannot have any arguments.

Appendix B: USER-DEFINED LIST STRUCTURES •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level list nesting in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

```
A.
   [1]
      •
         a)
            +
```

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register *:g* is used by the PWB/MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments *:g*, and each .LE call decrements it.

```
.de aL
'\"       register g is used as a local temporary to save :g before it is changed below
.nr g \\n(:g
.if \\ng=0 .AL A \" give me an A.
.if \\ng=1 .LB \\n(Li 0 1 4 \" give me a [1]
.if \\ng=2 .BL \" give me a bullet
.if \\ng=3 .LB \\n(Li 0 2 2 a \" give me an a)
.if \\ng=4 .ML + \" give me a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE
```

will yield:

```
   A. first line.
      [1] second line.
   B. third line.
```

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item:

```
.de bL
.ie \\n(.$ .nr g \\$1 \" if there is an argument, that is the level
.el .nr g \\n(:g \" if no argument, use current level
.if \\ng-\\n(:g>1 .)D "••ILLEGAL SKIPPING OF LEVEL" \" increasing level by more than 1
.if \\ng>\\n(:g \{.aL \\ng-1 \" if g > :g, begin new list
.         nr g \\n(:g\} \" and reset g to current level (.aL changes g)
.if \\n(:g>\\ng .LC \\ng \" if :g > g, prune back to correct level
'\"       if :g = g, stay within current list
.LI \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of *g* from its argument, rather than from *:g*. Invoking .bL without arguments causes it to stay at the current list level. The PWB/MM .LC macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call ".LC 0". If text is to be resumed at the end of a list, insert the call ".LC 0" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

The quick brown fox jumped over the lazy dog's back.

A. first line.

[1] second line.

B. third line.

C. fourth line.

fifth line.

Appendix C: SAMPLE FOOTNOTES

The following example illustrates several footnote styles and both labeled and automatically-numbered footnotes. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page:

With the footnote style set to the *nroff* default, we process a footnote[1] followed by another one.***** Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,[2] and another.† The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.[3]

---

1. This is the first footnote text example (.FD 10). This is the default style for *nroff.* The right margin is *not* justified. Hyphenation is *not* permitted. The text is indented, and the automatically generated label is *right*-justified in the text-indent space.

***** This is the second footnote text example (.FD 10). This is also the default *nroff* style but with a long footnote label provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is *left*-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is *not* justified, hyphenation is *not* permitted, the footnote text is *not* indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 10
With the footnote style set to the
.I nroff
default, we process a footnote\*F
.FS
This is the first footnote text example (.FD 10).
This is the default style for
.I nroff.
The right margin is
.I not
justified.
Hyphenation is
.I not
permitted.
The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.*****\□                                    (□ stands for a space)
.FS *****
This is the second footnote text example (.FD 10).
This is also the default
.I nroff
style but with a long footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to hyphenate, right margin justification,
indent, and left justify the label.
Here is a footnote,\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified, the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example, hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\□
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification,
no indentation, and with the label left-justified.
Here comes the final one.\*F\□
.FS
This is the fifth footnote example (.FD 6).
The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid of their country.
.FE
```

Appendix D: SAMPLE LETTER

☛ *The* nroff *and* troff *outputs corresponding to the input text below are shown on the following pages.*

```
.ND "November 1, 1977"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the Out-of-Hours course
"Document Preparation on the PWB/UNIX•
.FS •
UNIX is a Trademark of Bell Laboratories.
.FE
time-sharing system":
.P
The course is intended for clerks, typists, and others
who intend to use the PWB/UNIX system
for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system;
using appropriate output terminals.
.LI Files:
how text is stored on the system;
directories;
manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier to make;
how to use the editing system to
add, delete, and move lines of text;
how to make corrections.
.LI "Text processing:"
basic concepts;
use of general-purpose formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist such as the
.I "typo, spell, diff,"
and
.I grep
commands and a desk-calculator package.
.LE
.SG jrm
.NS
S. P. Lename
H. O. Del
M. Hill
.NE
```

Bell Laboratories

subject: Out-of-Hours Course Description
Case: 334455

date: November 1, 1977

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

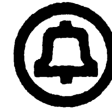
Other facilities: additional capabilities useful to the typist such as the typo, spell, diff, and grep commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.



Bell Laboratories

subject: **Out-of-Hours Course Description**
Case: **334455**

date: **November 1, 1977**

from: **D. W. Stevenson**
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the *typo*, *spell*, *diff*, and *grep* commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.

Appendix E: ERROR MESSAGES

I. PWB/MM Error Messages

Each PWB/MM error message consists of a standard part followed by a variable part. The standard part is of the form:

ERROR:input line *n*:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:[23]

Message	Explanation
Check TL, AU, AS, AE, MT sequence	The proper sequence of macros for the beginning of a memorandum is shown in {6.9}. Something has disturbed this order.
AL:bad arg:value	The argument to the .AL macro is not one of 1, A, a, I, or i. The incorrect argument is shown as *value*.
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.4}.
DS:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DS:missing DE	.DS or .DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
FE:no FS	.FE has been encountered with no previous .FS to match it.
FS:illegal inside TL or AS	.FS-.FE pair cannot be used inside the memorandum title or abstract.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but *value* has been supplied instead.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing arg	.H needs at least 1 argument.
HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.

23. This list is set up by ".LB 37 0 2 0" {5.4}.

LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as *value*.
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.

II. Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which the *user* has (some) control are listed below. Any other error messages should be reported to the local system-support group.

"Cannot open *filename*" is issued if one of the files in the list of files to be processed cannot be opened. If the filename is of the form /usr/lib/tmac.*name*, then the option -m*name* specifies an incorrect *name*. If the filename is of the form /usr/lib/term/*name*, then the *nroff* option -T*name* is incorrect. If the filename is of the form /usr/lib/font/*xx*, then the font specified in a formatter .fp request is incorrect.

"Exception word list full" indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

"Line overflow" means that the output line being generated was too long for the formatter's line buffer. The excess was discarded. See the "Word overflow" message below.

"Out of temp file space" means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.

"Too many page numbers" is issued when the list of pages specified to the formatter *-o* option is too long.

"Too many string/macro names" is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.

"Too many number registers" means that the pool of number register names is full. Unneeded registers can be deleted by using the .rr request.

"Word overflow" means that a word being generated exceeded the formatter's word buffer. The excess characters were discarded. A likely cause for this and for the "Line overflow" message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by *eqn*(I)/*neqn*(I).

Appendix F: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

1. Macros

The following is an alphabetical list of macro names used by PWB/MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are "user exits" called from inside header, footer, or other macros.

1C One-column processing (11.4)
.1C

2C Two-column processing (11.4)
.2C

AE Abstract end (6.4)
.AE

AF Alternate format of "Subject/Date/From" block (6.7.2)
.AF [company-name]

AL Automatically-incremented list start (5.3.3.1)
.AL [type] [text-indent] [1]

AS Abstract start (6.4)
.AS [arg] [indent]

AU Author information (6.2)
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

B Bold (underline in *nroff*) (11.1)
.B [bold-arg] [previous-font-arg]

BL Bullet list start (5.3.3.2)
.BL [text-indent] [1]

CS Cover sheet (10.2)
.CS [pages] [other] [total] [figs] [tbls] [refs]

DE Display end (7.1)
.DE

DF Display floating start (7.2)
.DF [format] [fill]

DL Dash list start (5.3.3.3)
.DL [text-indent] [1]

DS Display static start (7.1)
.DS [format] [fill]

EC Equation caption (7.5)
.EC [title] [override] [flag]

EF Even-page footer (9.6)
.EF [arg]

EH Even-page header (9.3)
.EH [arg]

EN End equation display (7.4)
.EN

EQ Equation display start (7.4)
.EQ

FD Footnote default format (8.3)
.FD [arg] [1]

FE Footnote end (8.2)
.FE

FG Figure title (7.5)
.FG [title] [override] [flag]

FS Footnote start (8.2)
.FS [label]

H Heading—numbered (4.2)
.H level [heading-text]

HC Hyphenation character (3.4)
.HC [hyphenation-indicator]

HM Heading mark style (Arabic or Roman numerals, or letters) (4.2.2.5)
.HM [arg1] ... [arg7]

HU Heading—unnumbered (4.3)
.HU heading-text

HX * Heading user exit X (before printing heading) (4.6)
.HX dlevel rlevel heading-text

HZ * Heading user exit Z (after printing heading) (4.6)
.HZ dlevel rlevel heading-text

I Italic (underline in *nroff*) (11.1)
.I [italic-arg] [previous-font-arg]

LB List begin (5.4)
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

LC List-status clear (Appendix B)
.LC [list-level]

LE List end (5.3.2)
.LE [1]

LI List item (5.3.1)
.LI [mark] [1]

ML Marked list start (5.3.3.4)
.ML mark [text-indent] [1]

MT Memorandum type (6.6)
.MT [type] [1]

ND New date (6.7.1)
.ND new-date

NE Notation end (6.11.2)
.NE

NS Notation start (6.11.2)
.NS [arg]

OF Odd-page footer (9.7)
.OF [arg]

OH Odd-page header (9.4)
.OH [arg]

OK Other keywords for TM cover sheet (6.5)
.OK [keyword] ...

P Paragraph {4.1}
.P [type]

PF Page footer {9.5}
.PF [arg]

PH Page header {9.2}
.PH [arg]

PX * Page-header user exit {9.12}
.PX

R Return to regular (roman) font (end underlining in *nroff*) {11.1}
.R

RL Reference list start {5.3.3.5}
.RL [text-indent] [1]

S Set *troff* point size and vertical spacing {11.8}
.S [arg]

SA Set adjustment (right-margin justification) default {11.2}
.SA [arg]

SG Signature line {6.11.1}
.SG [arg] [1]

SK Skip pages {11.7}
.SK [pages]

SP Space—vertically {11.6}
.SP [lines]

TB Table title {7.5}
.TB [title] [override] [flag]

TC Table of contents {10.1}
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

TE Table end {7.3}
.TE

TL Title of memorandum {6.1}
.TL [charging-case] [filing-case]

TM Technical Memorandum number(s) {6.3}
.TM [number] ...

TP * Top-of-page macro {9.12}
.TP

TS Table start {7.3}
.TS

TX * Table-of-contents user exit {10.1}
.TX

VL Variable-item list start {5.3.3.6}
.VL text-indent [mark-indent] [1]

II. Strings

The following is an alphabetical list of string names used by PWB/MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU Bullet {3.7}
nroff: ⊕
troff: •

F Footnote numberer {8.1}
nroff: \u\\n+ (:p\d
troff: \v'-.4m'\s-3\\n+ (:p\s0\v'.4m'

DT Date (current date, unless overridden) {6.7.1}
Month day, year (e.g., October 31, 1977)

HF Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1}
3 3 2 2 2 2 2 (all underlined in *nroff*, and B B I I I I I in *troff*)

RE SCCS Release and Level of PWB/MM {11.3}
Release.Level (e.g., 12.2)

Note that if the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}.

III. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the PWB/MM macro definitions are read by the formatter {2.4, 2.5}. See {1.4} for notes on setting and referencing registers.

A * Has the effect of invoking the .AF macro without an argument {2.4}
0, [0:1]

Au Inhibits printing of author's location, department, room, and extension in the "from" portion of a memorandum {6.2}
1, [0:1]

B * Defines table-of-contents and/or cover-sheet macros {2.4}
0, [0:3]

C * Copy type (Original, DRAFT, etc.) {2.4}
0 (Original), [0:3]

Cl Contents level (i.e., level of headings saved for table of contents) {4.4}
2, [0:7]

D * Debug flag {2.4}
0, [0:1]

Ds Static display pre- and post-space {7.1}
1, [0:1]

Ec Equation counter, used by .EC macro {7.5}
0, [0:?], incremented by 1 for each .EC call.

Ej Page-ejection flag for headings {4.2.2.1}
0 (no eject), [0:7]

Fg Figure counter, used by .FG macro {7.5}
0, [0:?], incremented by 1 for each .FG call.

Fs Footnote space (i.e., spacing between footnotes) {8.4}
1, [0:?]

H1-H7 Heading counters for levels 1-7 {4.2.2.5}
0, [0:?], incremented by .H of corresponding level or .HU if at level given by register *Hu*. H2-H7 are reset to 0 by any heading at a lower-numbered level.

Hb Heading break level (after .H and .HU) {4.2.2.2}
2, [0:7]

Hc Heading centering level for .H and .HU {4.2.2.3}
0 (no centered headings), [0:7]

Hi Heading temporary indent (after .H and .HU) {4.2.2.2}
1 (indent as paragraph), [0:2]

Hs Heading space level (after .H and .HU) {4.2.2.2}
2 (space only after .H 1 and .H 2), [0:7]

Ht Heading type (for .H: single or concatenated numbers) {4.2.2.5}
0 (concatenated numbers: 1.1.1, etc.), [0:1]

Hu Heading level for unnumbered heading (.HU) {4.3}
2 (.HU at the same level as .H 2), [0:7]

Hy Hyphenation control for body of document {3.4}
1 (automatic hyphenation on), [0:1]

L * Length of page {2.4}
66, [20:?] (11i, [2i:?] in *troff*)[24]

Li List indent {5.3.3.1}
5, [0:?]

N * Numbering style {2.4}
0, [0:3]

O * Offset of page {2.4}
0, [0:?] (0.5i, [0i:?] in *troff*)[24]

P Page number, managed by PWB/MM {2.4}
0, [0:?]

Pi Paragraph indent {4.1}
5, [0:?]

Pt Paragraph type {4.1}
2 (paragraphs indented except after headings, lists, and displays), [0:2]

S * *Troff* default point size {2.4}
10, [6:36]

Si Standard indent for displays {7.1}
5, [0:?]

T * Type of *nroff* output device {2.4}
0, [0:2]

Tb Table counter {7.5}
0, [0:?], incremented by 1 for each .TB call.

U * Underlining style (*nroff*) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]

W * Width of page (line and title length) {2.4}
65, [10:1365] (6.5i, [2i:7.54i] in *troff*)[24]

---

24. For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.

Typing Documents with PWB/MM

D. W. Smith and E. M. Piskorik

Bell Laboratories
Piscataway, New Jersey 08854

This guide shows several examples of documents prepared with PWB/MM, a set of general-purpose formatting macros used with the PWB/UNIX* text formatters *nroff* and *troff* (as well as with the *eqn/neqn* and *tbl* programs) to produce memoranda, letters, books, manuals, etc. References to manuals for these programs are given on p. 16.

In the examples, input is shown in this
    Helvetica sans serif font.

The resulting output is shown (boxed) in this
    Times Roman font.

Substitutable arguments are shown in this
    Times Roman Italic font.

Square brackets (*[...]*) indicate that the enclosed substitutable argument is optional.

All output shown in the examples was done by *troff*; *nroff* output would look somewhat different.†

Contents

Paragraphs and Headings 2
Paragraph and Heading Parameters 2
Lists and List Types 4
Nested Lists 5
Italic, Bold, and Underlining 5
Displays 6
Footnotes 6
Simple Letter—Example 7
Technical Memorandum—Example 9
Memorandum-Style Macros 11
Two-Column Output 13
Equations 14
Tables 15
How to Get Output 16
References 16

* UNIX is a Trademark of Bell Laboratories.

† For example, what we call a "blank line" is a blank line in *nroff*, but is ½ of a vertical space in *troff*, while headings that are underlined in *nroff* are either **bold** or *italic* in *troff*.

Paragraphs and Headings

☛ *The output for the following is shown on p. 3.*

```
.H 1 "PARAGRAPHS AND HEADINGS"
This section describes the types of paragraphs
and the kinds of headings that are available.
.H 2 Paragraphs
Paragraphs are specified by the .P macro.
Usually, they are indented except
after headings, lists, and displays.
The number register Pt is used
to change the paragraph style.
.H 2 Headings
.H 3 "Numbered Headings."
There are seven levels of numbered headings.
Level 1 is the most major or highest;
level 7, the lowest.
.P
Headings are specified with the .H macro, whose
first argument is the level of heading (1 through 7).
.P
The appearance of headings varies according to
the level.
On output, level 1 headings are preceded by two
blank lines; all others are preceded by one blank line.
Level 1 and level 2 headings produce stand-alone
headings, underlined in
.I nroff
and bold in
.I troff.
Levels 3 through 7 are run-in and underlined (or italic).
.H 3 "Unnumbered Headings."
The macro .HU is a special case
of .H, in that no heading number is printed.
Each .HU heading has the level given by
the register Hu, whose initial value is 2.
Usually, the value of that register is
set to make unnumbered headings (if any) occur
at the lowest heading level in a document.
```

Paragraph and Heading Parameters

There are many parameters that can change the output appearance of headings and paragraphs. Given below are some of these parameters, their *default* values, and their meanings (level 1 is the *most major* or *highest*, while level 7 is the *lowest*):

.nr Pi 5	paragraph-indent in characters (or ens).
.nr Pt 0	never indent paragraphs.
.nr Pt 1	always indent paragraphs.
.nr Pt 2	indent paragraphs *except* after headings, lists, and displays (*default*).
.ds HF 3 3 2 2 2 2 2	font specification for each of the 7 heading levels: 1 indicates roman, 2 indicates italic, 3 indicates bold.

3

-1-

1. PARAGRAPHS AND HEADINGS

This section describes the types of paragraphs and the kinds of headings that are available.

1.1 Paragraphs

Paragraphs are specified by the .P macro. Usually, they are indented except after headings, lists, and displays. The number register Pt is used to change the paragraph style.

1.2 Headings

1.2.1 Numbered Headings. There are seven levels of numbered headings. Level 1 is the most major or highest; level 7, the lowest.

Headings are specified with the .H macro, whose first argument is the level of heading (1 through 7).

The appearance of headings varies according to the level. On output, level 1 headings are preceded by two blank lines; all others are preceded by one blank line. Level 1 and level 2 headings produce stand-alone headings, underlined in *nroff* and bold in *troff*. Levels 3 through 7 are run-in and underlined (or italic).

1.2.2 Unnumbered Headings. The macro .HU is a special case of .H, in that no heading number is printed. Each .HU heading has the level given by the register Hu, whose initial value is 2. Usually, the value of that register is set to make unnumbered headings (if any) occur at the lowest heading level in a document.

.HM 1 1 1 1 1 1 1
"marking" style for each heading level; the above yields an all-numeric marking style. Available styles are: 1, 0001, A, a, I, and i.

.nr Hb 2 lowest heading level that is stand-alone (i. e., *not* run-in with the following text).

.nr Hc 0 lowest heading level that is centered.

.nr Hs 2 lowest heading level after which there is a blank line.

.nr Ht 0 heading marks will be concatenated.

.nr Hu 2 unnumbered headings (.HU) are equivalent to numbered headings at this level for spacing, font, and counting.

.nr Cl 2 lowest heading level to be saved for the table of contents.

.nr Ej 0 lowest heading level that forces the start of a new page.

4

Default Heading Style	
to get:	*type:*
n. HEADING Text ...	.H 1 "HEADING" Text ...
n.n Heading Text ...	.H 2 "Heading" Text ...
n.n.n Heading. Text ...	.H 3 "Heading." Text ...

Lists and List Types

All lists have a *list begin* macro, one or more *list items*—each consisting of a .LI macro followed by the *list item text*—and the *list end* macro .LE. That is, lists are typed like this:

list begin macro
.LI
list item text ...
.LI
list item text ...
⋮
.LE

where the *list begin macro* is one of the following:

.AL *[type] [indent]*	automatic list (*type* is 1, A, a, I, or i; if omitted, defaults to 1)
.BL *[indent]*	bullet list
.DL *[indent]*	dash list
.ML *mark [indent]*	marked list (*mark* is the desired mark)
.RL *[indent]*	reference list
.VL *indent*	variable list

indent is the number of characters of indentation (from the current indent) at which the list is to start; if it is optional and omitted, the default indentation for the given list style is used; *mark* will appear to the left of the indentation.

☛ *The output for the following is shown on p. 5.*

```
.AL 1
.LI
Pencilpusher, I., and Hardwired, X.
A New Kind of Set Screw.
.I "Proc. IEEE"
.B 75
(1976), 235-41.
.LI
Nails, H., and Irons, R.
Fasteners for Printed Circuit Boards.
.I "Proc. ASME"
.B 123
(1974), 23-24.
.LE
```

1. Pencilpusher, I., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* **75** (1976), 235-41.
2. Nails, H., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* **123** (1974), 23-24.

Nested Lists

```
This is ordinary text to show
the margins of the page.
.AL 1
.LI
First-level item.
.AL a
.LI
Second-level item.
.LI
Another second-level item, but
somewhat longer.
.LE
.LI
Return to previous list (and to previous value
of indentation) at this point.
.LI
Another line.
.LE
.P
Now we're out of the lists and at the margin that
existed at the beginning of this example.
```

This is ordinary text to show the margins of the page.

1. First-level item.
   a. Second-level item.
   b. Another second-level item, but somewhat longer.
2. Return to previous list (and to previous value of indentation) at this point.
3. Another line.

Now we're out of the lists and at the margin that existed at the beginning of this example.

Italic, Bold, and Underlining

In the examples on pp. 4 and 7, the macros .I, .B, and .R are used to change to, respectively, the italic, bold, and roman fonts in *troff*. In *nroff*, both .I and .B cause underlining until the occurrence of .R, which turns it off. A single argument given to either .I or .B results in that argument being underlined by *nroff*, or printed in the corresponding font by *troff*.

Displays

Displays are blocks of text that are to be kept together—not split across pages. A static display (.DS) appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of a page if a static display is too big to fit there. A floating display (.DF), on the other hand, will "float" through the input text to the top of the next page if there is not enough room for it on the current page; thus, the text that *follows* a floating display in the input may *precede* it in the output. Displays can be positioned at the left margin, indented, or centered.

```
.DS [format] [fill]        .DF [format] [fill]
text ...                   text ...
.DE                        .DE
```

where *format* and *fill* have the following meanings:

format		*fill*	
Code	*Meaning*	*Code*	*Meaning*
""	no indent	""	no fill
0	no indent	0	no fill
1	indent	1	fill
2	center		

```
Highland Avenue, Mountain Station,
South Orange, Maplewood, Millburn, Short Hills;
.DS 1
and now
for something
completely different
.DE
Summit, Chatham, Madison,
Convent Station, Morristown, New Providence,
Murray Hill, Berkeley Heights.
```

Highland Avenue, Mountain Station, South Orange, Maplewood, Millburn, Short Hills;

    and now
    for something
    completely different

Summit, Chatham, Madison, Convent Station, Morristown, New Providence, Murray Hill, Berkeley Heights.

Footnotes

Two styles of footnote marking are shown on p. 7. In the first, the asterisk is the mark placed on the footnote and the following .FS macro call, while in the second, a number is *automatically* generated to mark the footnote. The macros .FS and .FE are used to delimit the footnote text that is to appear at the bottom of the page.

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without this basic tool,\*
.FS *
As first shown by Tiger & Leopard (1975).
.FE
few assemblies could be completed.
They may lack the popular\*F
.FS
According to Panther & Lion (1977).
.FE
appeal of the sledgehammer . . .
```

> Among the most important occupants of the workbench are the long-nosed pliers. Without this basic tool,* few assemblies could be completed. They may lack the popular[1] appeal of the sledgehammer ...
>
> * As first shown by Tiger & Leopard (1975).
> 1. According to Panther & Lion (1977).

Simple Letter—Example

☛ *The output for the following is shown on p. 8.*

```
.nr Pt 0
.ND "May 1, 1977"
.TL
PWB/MM Class
.AU "J. J. Jones" JJJ PY 9999 5001 1Q-100
.MT ""
.DS
To All Students:
.DE
.P
There will be a class on the document preparation
facilities of PWB/MM on November 15-18.
This class lasts for 4 half-day (morning) sessions,
each consisting of a lecture
and practice exercises on the system.
.P
The meeting rooms for the class are:
.DS 1
.ta 15n                (n represents character positions)
Monday→4D-502                        (→ indicates a tab)
Tuesday→4D-502
Wednesday→2B-639
Thursday→2C-641.
.DE
.P
Please read the following before attending class:
.DL
.LI
.I "UNIX for Beginners,"
Sections I and II.
.LI
.I
A Tutorial Introduction to the UNIX Text Editor.
.R
.LE
```

(input example continued on the next page)

Bell Laboratories

subject: **PWB/MM Class** date: **May 1, 1977**

from: **J. J. Jones**
PY 9999
1Q-100 x5001

To All Students:

There will be a class on the document preparation facilities of PWB/MM on November 15-18. This class lasts for 4 half-day (morning) sessions, each consisting of a lecture and practice exercises on the system.

The meeting rooms for the class are:

Monday	4D-502
Tuesday	4D-502
Wednesday	2B-639
Thursday	2C-641.

Please read the following before attending class:

— *UNIX for Beginners*, Sections I and II.

— *A Tutorial Introduction to the UNIX Text Editor.*

These can be obtained from the Computing Information Library.

PY-9999-JJJ-ae **J. J. Jones**

Copy to
G. H. Hurtz
S. P. LeName

(input example continued from the previous page)

```
.P
These can be obtained from the Computing
Information Library.
.SG ae
.NS
G. H. Hurtz
S. P. LeName
.NE
```

Technical Memorandum—Example

☛ *The output for the following is shown on pp. 10-12.*

```
.nr Pt 1
.ND "June 29, 1977"
.TL 12345 666666
On Constructing a Table of All
Even Prime Numbers
.AU "S. P. LeName" SPL PY 9999 4000 1Z-123
.AU "G. H. Hurtz" GHH PY 9999 4001 1Z-121
.TM 76543210
.AS
.P
This is an abstract for a technical memorandum.
The abstract will appear on the cover
sheet and on the first page
.I unless
the macro .AS has an argument of 1, in which case
the abstract will be printed only on the cover sheet.
The TM number appears on the cover sheet
and on the first page.
"Other Keywords" appear only on the cover sheet.
.P
The abstract may consist of one or more paragraphs;
it must fit on the cover sheet.
.AE
.OK "Prime Numbers" Even
.MT
.H 1 "INTRODUCTORY MATERIAL"
The first line of the body of the memorandum
immediately follows the macro call for
the heading (.H).
Alternately, lower-level heading macros may follow it,
as well as macros for lists, paragraphs, and so on.
A brief example of a list follows:
.AL A
.LI
This is the first item in an alphabetical
list in the body of this memorandum.
.LI
This is the second item in the list.
.AL 1
.LI
This is the first item in a (numbered) sub-list.
.LI
This is the second item in that sub-list.
.LE
.LE
.P
This is the second paragraph under the first heading.
In addition to alphabetized and numbered lists, there
are bullet lists, dash lists, variable lists, etc.
.H 2 "First Second-Level Heading"
This is the first paragraph under a
second-level heading.
Notice how that heading is numbered and
where the heading and text are printed.
.H 1 "SECOND FIRST-LEVEL HEADING"
This is the first paragraph under the
second first-level heading of the memorandum.
```

(input example continued on the next page)

Bell Laboratories

subject: **On Constructing a Table of All Even Prime Numbers**
Case: **12345**
File: **666666**

date: **June 29, 1977**

from: **S. P. LeName**
PY 9999
1Z-123 x4000
G. H. Hurtz
PY 9999
1Z-121 x4001

TM: **76543210**

ABSTRACT

This is an abstract for a technical memorandum. The abstract will appear on the cover sheet and on the first page *unless* the macro *.AS* has an argument of 1, in which case the abstract will be printed only on the cover sheet. The TM number appears on the cover sheet and on the first page. "Other Keywords" appear only on the cover sheet.

The abstract may consist of one or more paragraphs; it must fit on the cover sheet.

MEMORANDUM FOR FILE

1. INTRODUCTORY MATERIAL

The first line of the body of the memorandum immediately follows the macro call for the heading (.H). Alternately, lower-level heading macros may follow

---

(input example continued from the previous page)

```
.HU REFERENCES
.RL
.LI
Pencilpusher, I., and Hardwired, X.
A New Kind of Set Screw.
.I "Proc. IEEE"
.B 75
(1976), 235-41.
.LI
Nails, H., and Irons, R.
Fasteners for Printed Circuit Boards.
.I "Proc. ASME"
.B 123
(1974), 23-24.
.LE
.SG rfg
.NS 3
.NS 2
G. B. Brown
C. P. Jones
J. J. Smith
.NE
.CS 2 1 3 0 0 2
```

11

-2-

it, as well as macros for lists, paragraphs, and so on. A brief example of a list follows:

A. This is the first item in an alphabetical list in the body of this memorandum.

B. This is the second item in the list.

   1. This is the first item in a (numbered) sub-list.

   2. This is the second item in that sub-list.

This is the second paragraph under the first heading. In addition to alphabetized and numbered lists, there are bullet lists, dash lists, variable lists, etc.

1.1 **First Second-Level Heading**

This is the first paragraph under a second-level heading. Notice how that heading is numbered and where the heading and text are printed.

2. **SECOND FIRST-LEVEL HEADING**

This is the first paragraph under the second first-level heading of the memorandum.

REFERENCES

[1] Pencilpusher, I., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* 75 (1976), 235-41.

[2] Nails, H., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* 123 (1974), 23-24.

S. P. LeName

PY-9999-SPL/GHH-rfg **G. H. Hurtz**

Att.

Copy (without att.) to
G. B. Brown
C. P. Jones
J. J. Smith

12

Bell Laboratories Cover Sheet for TM

The information contained herein ... not for publication ...

Title: **On Constructing a Table of All Even Prime Numbers** Date: **June 29, 1977**

TM: **76543210**

Other Keywords: **Prime Numbers**
Even

Author(s)	Location	Ext.	
S. P. LeName	**PY 1Z-123**	**4000**	Charging Case: **12345**
G. H. Hurtz	**PY 1Z-121**	**4001**	Filing Case: **666666**

ABSTRACT

This is an abstract for a technical memorandum. The abstract will appear on the cover sheet and on the first page *unless* the macro .AS has an argument of 1, in which case the abstract will be printed only on the cover sheet. The TM number appears on the cover sheet and on the first page. "Other Keywords" appear only on the cover sheet.

The abstract may consist of one or more paragraphs; it must fit on the cover sheet.

Pages Text: 2 Other: 1 Total: 3

No. Figures: 0 No. Tables: 0 No. Refs.: 2

Z-0000-X SEE REVERSE SIDE FOR DISTRIBUTION LIST

Memorandum-Style Macros

Macros for a memorandum-style document must be invoked in the order shown on pp. 9-10. Once the "memorandum type" (.MT) macro has been invoked, none of the macros that precede it can be used. The .MT macro controls the format of the "subject, date, from" portion of the first page of the memorandum. Different arguments to the .MT macro will produce different kinds of memoranda:

Code	*Meaning*
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-Paper style
.MT 5	External Letter

The input and the resulting output for a simple letter are shown on pp. 7-8. Note that the .TM, .AS/.AE, and .OK macros are *not* used there, and that the .MT macro has a *null* argument (""). Documents of the type shown on pages 2-3 (essentially plain text) are produced by omitting, as well, the other "memorandum-style" macros: .ND, .TL, .AU, and .MT at the beginning of the document, and .SG, .NS/.NE, and .CS at the end.

Like the .MT macro, the notation macro (.NS) may also take different arguments to produce a variety of notations following the signature line:

Code	*Meaning*
.NS ""	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to

If the .CS macro is included in the input file (see last line of p. 10) *and* if the -rB2 option is included on the command line (see p. 16), a cover sheet is generated (see p. 12). (The 6 arguments to .CS are the data for the bottom of the TM cover sheet: "Pages Text," "Other," etc.) Similarly, the .TC macro, together with the -rB1 or -rB3 option (see p. 16) generates a table of contents; .CS and .TC can occur only at the end of a document.

Two-Column Output

```
.DS 2
The Declaration of Independence
.DE
.2C
.P
When in the Course of human events, it becomes
necessary for one people to dissolve the political
bands which have connected them with another, and
to assume among the powers of the earth, the
separate and equal station to which the Laws of
Nature and of Nature's God entitle them, a decent
respect to the opinions of mankind requires that they
should declare the causes which impel them to the
separation.
.P
We hold these truths to be self-evident, that all men
are created equal, ...
```

The Declaration of Independence

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, ...

Equations

A stand-alone equation is built within a display.

```
.DS 2
.EQ
x sup 2 over a sup 2 ~=~ sqrt { pz sup 2 + qz + r }
.EN
.DE
```

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r}$$

```
.DS 1
.EQ
bold V bar sub nu~=~left [ pile { a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above . } col { . above .
above A(33) } } right ] times left [ pile { alpha
above beta above gamma } right ]
.EN
.DE
```

$$\bar{\mathbf{V}}_\nu = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \times \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

In-line equations may appear in running text if a character has been defined to mark the left and right ends of the equation. Normally, $ is used as that character and is so defined by typing the following three lines at the beginning of the document:

```
.EQ
delim $$
.EN
```

```
The quantities $a dot$, $b dotdot$, $xi tilde times
y vec$ are the values that show ...
```

The quantities $\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$ are the values that show ...

This facility can be used for preparing text that contains subscripts and superscripts:

```
The quantity $ a sub j sup 3 $ is ...
```

The quantity a_j^3 is ...

For more examples, see p. 15 and Reference 4.

15

Tables

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	alphabetic subcolumn
l	left-adjust	s	spanned

Global table options are *center, expand, box, allbox, doublebox,* and *tab (x).*

```
.DS
.TS
allbox ;
ci s s
c c c
n n n .
AT&T Common Stock
Year—Price—Dividend
1973—46-55—2.87
4—40-53—3.24
5—45-52—3.40
6—51-59—.95•
.TE
• (first quarter only)
.DE
```

AT&T Common Stock		
Year	Price	Dividend
1973	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

(— indicates a tab)

```
.EQ
delim $$
.EN
.DS
.TS
box ;
cf2 cf2
l l .
Name—Definition
_
.sp
Sine—$sin ( x ) = 1 over 2j ( e sup jx - e sup -jx )$
Zeta—$zeta ( s ) = \
sum from k=1 to inf k sup -s ~~( Re~s > 1 )$
.TE
.DE
```

Name	*Definition*
Sine	$\sin(x)=\frac{1}{2j}(e^{jx}-e^{-jx})$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s} \quad (\mathrm{Re}\ s>1)$

For more examples, see Reference 3.

16

How to Get Output

Documents with text only:

nroff: mm *[options]* files
or nroff *[options]* -mm files
troff: troff *[options]* -mm files

Text and equations:

nroff: mm -e *[options]* files
or neqn files | nroff *[options]* -mm -
troff: eqn files | troff *[options]* -mm -

Text and tables:

nroff: mm -t *[options]* files
or tbl files | nroff -mm *[options]* -
troff: tbl files | troff -mm *[options]* -

Text, tables, and equations:

nroff: mm -t -e *[options]* files
or tbl files | neqn | nroff *[options]* -mm -
troff: tbl files | eqn | troff *[options]* -mm -

The following options may be specified on the above PWB/UNIX shell command lines:

-o*k*,*m*-*n*	print only page *k*, and pages *m* through *n*.
-rB1	include macros for the table of contents.
-rB2	include macros for the cover sheet.
-rB3	include macros for both.
-rC1	*OFFICIAL FILE COPY* in footer.
-rC2	*DATE FILE COPY* in footer.
-rC3	*DRAFT* in footer.
-rL*n*	set page length to *n* lines.*
-rN1	page header at *bottom* of first page only.
-rN2	no page number on first page.
-rN3	*section-page* numbering.
-rO*n*	set page offset to *n* characters.*
-rW*n*	set line width to *n* characters.*

Terminal type and/or pitch are usually indicated by the -hp, -ti, -450, -300S, and/or -12 options of the *mm*(I) command, if it is used (see Reference 6); otherwise, they are specified by one of the *nroff* -T*name* options.

References

1. *PWB/MM—Programmer's Workbench Memorandum Macros* by D. W. Smith and J. R. Mashey.
2. *A Tutorial Introduction to the UNIX Text Editor* by B. W. Kernighan.
3. *Tbl—A Program to Format Tables* by M. E. Lesk.
4. *Typesetting Mathematics—User's Guide (Second Edition)* by B. W. Kernighan and L. L. Cherry.
5. *NROFF/TROFF User's Manual* by J. F. Ossanna.
6. *PWB/UNIX User's Manual—Edition 1.0* by T. A. Dolotta, R. C. Haight, and E. M. Piskorik, eds.

* For *nroff*, *n* must be an *unscaled* number representing lines or character positions. For *troff*, *n* must be *scaled*.

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

January 16, 1979

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] or various layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as "*troff* or *nroff*."

The input to *tbl* is text for a document, with tables preceded by a ".TS" (table start) command and followed by a ".TE" (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The ".TS" and ".TE" lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [4]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the ".TS" or ".TE" lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

text
.TS
table
.TE
text
.TS
table
.TE
text
. . .

where the format of each table is as follows:

.TS
options ;
format .
data
.TE

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

1) OPTIONS. There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

   center — center the table (default is left-adjust);

   expand — make the table as wide as the current line length;

   box — enclose the table in a box;

   allbox — enclose each item in the table in a box;

   doublebox — enclose the table in two boxes;

   tab (*x*) — use *x* instead of tab to separate data items.

   linesize (*n*) — set lines or rules (e.g. from **box**) in *n* point type;

   delim (*xy*) — recognize *x* and *y* as the *eqn* delimiters.

   The *tbl* program tries to keep boxed tables on one page by issuing appropriate "need" (*.ne*) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2) FORMAT. The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

   L or **l** to indicate a left-adjusted column entry;

   R or **r** to indicate a right-adjusted column entry;

   C or **c** to indicate a centered column entry;

   N or **n** to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;

   A or **a** to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);

   S or **s** to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or

   ^ to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

   When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as

shown on the right:

```
13              13
4.2              4.2
26.4.12         26.4.12
abc               abc
abc\&            abc
43\&3.22         433.22
749.12          749.12
```

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c s s
l n n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

Overall title		
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the "expand" option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter **f** or **F**. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

3) DATA. The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables — An input line beginning with a '.' followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ".sp" commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (underscore) or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table.*

Single column horizontal lines — An input table *entry* containing only the character _ or = is taken to be a single or double line extending the full width of the *column.* Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table *entry* containing only the string _ is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form **\R***x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks — In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus the sequence

> . . . **T{**
> *block of*
> *text*
> **T}** . . .

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the **T}** end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 10 for an illustration of included text blocks in a table. If more than twenty or thirty text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff,* and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the ".TS" macro) and any table format specifications of size, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: — Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ".TS" command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry \s+3\fIdata\fP\s0). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

4) ADDITIONAL COMMAND LINES. If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ".T&" (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 10 and 12. Using this procedure, each table line can be close to its corresponding format line.

Warning: it is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff −ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special −*TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are −*ms* and −*mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled.. Users must also beware of using equations in **n**-style columns; this is nearly

always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as "1245 $+- 16$" will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms *#x*, *x+*, *x*|, ^*x*, and *x*–, where *x* is any lower case letter. The names ##, #–, and #^ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl*. The symbol Ⓣ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
LanguageⓉAuthorsⓉRuns on

FortranⓉManyⓉAlmost anything
PL/1ⓉIBMⓉ360/370
CⓉBTLⓉ11/45,H6000,370
BLISSⓉCarnegie-MellonⓉPDP-10,11
IDSⓉHoneywellⓉH6000
PascalⓉStanfordⓉ370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c|c|c
l|l|n.
Major New York Bridges
=
BridgeⓉDesignerⓉLength
_
BrooklynⓉJ. A. RoeblingⓉ1595
ManhattanⓉG. LindenthalⓉ1470
WilliamsburgⓉL. L. BuckⓉ1600
_
QueensboroughⓉPalmer &Ⓣ1182
Ⓣ Hornbostel
_
ⓉⓉ1380
TriboroughⓉO. H. AmmannⓉ_
ⓉⓉ383
_
Bronx WhitestoneⓉO. H. AmmannⓉ2300
Throgs NeckⓉO. H. AmmannⓉ1800
_
George WashingtonⓉO. H. AmmannⓉ3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn Manhattan Williamsburg	J. A. Roebling G. Lindenthal L. L. Buck	1595 1470 1600
Queensborough	Palmer & Hornbostel	1182
		1380
Triborough	O. H. Ammann	
		383
Bronx Whitestone Throgs Neck	O. H. Ammann O. H. Ammann	2300 1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
ⓣStack
ⓣ_
1ⓣ46
ⓣ_
2ⓣ23
ⓣ_
3ⓣ15
ⓣ_
4ⓣ6.5
ⓣ_
5ⓣ2.1
ⓣ_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L |LB
L L _
L L L.
januaryⓣfebruaryⓣmarch
aprilⓣmay
juneⓣjulyⓣMonths
augustⓣseptember
octoberⓣnovemberⓣdecember
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c |c s s
c |c s s
c |c |c |c.
Foodⓣ Percent by Weight
\^ⓣ_
\^ⓣProteinⓣFatⓣCarbo-
\^ⓣ\^ⓣ\^ⓣhydrate
_
.T&
l |n |n |n.
Applesⓣ.4ⓣ.5ⓣ13.0
Halibutⓣ18.4ⓣ5.2ⓣ. . .
Lima beansⓣ7.5ⓣ.8ⓣ22.0
Milkⓣ3.3ⓣ4.0ⓣ5.0
Mushroomsⓣ3.5ⓣ.4ⓣ6.0
Rye breadⓣ9.0ⓣ.6ⓣ52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI s s
c cw(1i) cw(1i)
lp9 lp9 lp9.
New York Area Rocks
EraⓣFormationⓣAge (years)
PrecambrianⓣReading Prongⓣ>1 billion
PaleozoicⓣManhattan Prongⓣ400 million
MesozoicⓣT{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T}ⓣ200 million
CenozoicⓣCoastal PlainⓣT{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
doublebox;
c c
l l.
NameⓉDefinition
.sp
.vs +2p
GammaⓉ$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty}k^{-s}\quad(\mathrm{Re}\,s>1)$

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c || c | c | c | c
c || c | c | c | c
r2 || n2 | n2 | n2 | n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line : Set : 1-Point : 2-Point : 4-Point
Width : Solid : Leading : Leading : Leading
_
9 Pica : \-9.3 : \-6.0 : \-5.3 : \-7.1
14 Pica : \-4.5 : \-0.6 : \-0.3 : \-1.7
19 Pica : \-5.0 : \-5.1 : 0.0 : \-2.0
31 Pica : \-3.7 : \-3.8 : \-2.4 : \-3.6
43 Pica : \-9.1 : \-9.0 : \-5.9 : \-8.8
.TE
```

Output:

Readability of Text
Line Width and Leading for 10-Point Type

Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route milesⓉ244
TubeⓉ66
Sub-surfaceⓉ22
SurfaceⓉ156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
JourneysⓉ674 million
Average lengthⓉ4.55 miles
Passenger milesⓉ3,066 million
.T&
l r
a r.
Passenger traffic \- road
JourneysⓉ2,252 million
Average lengthⓉ2.26 miles
Passenger milesⓉ5,094 million
.T&
l n
a n.
.sp .5
VehiclesⓉ12,521
Railway motor carsⓉ2,905
Railway trailer carsⓉ1,269
Total railwayⓉ4,174
OmnibusesⓉ8,347
.T&
l n
a n.
.sp .5
StaffⓉ73,739
Administrative, etc.Ⓣ5,582
Civil engineeringⓉ5,134
Electrical eng.Ⓣ1,714
Mech. eng. \- railwayⓉ4,310
Mech. eng. \- roadⓉ9,152
Railway operationsⓉ8,930
Road operationsⓉ35,946
OtherⓉ2,971
.TE
```

Output:

Some London Transport Statistics	
(Year 1964)	
Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic – railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic – road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. – railway	4,310
Mech. eng. – road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
Nameⓣ Office addressⓣ Phone
.sp .5
James J. Florioⓣ 23 S. White Horse Pike, Somerdale 08083ⓣ 609-627-8222
William J. Hughesⓣ 2920 Atlantic Ave., Atlantic City 08401ⓣ 609-345-4844
James J. Howardⓣ 801 Bangs Ave., Asbury Park 07712ⓣ 201-774-1600
Frank Thompson, Jr.ⓣ 10 Rutgers Pl., Trenton 08618ⓣ 609-599-1619
Andrew Maguireⓣ 115 W. Passaic St., Rochelle Park 07662ⓣ 201-843-0240
Robert A. Roeⓣ U.S.P.O., 194 Ward St., Paterson 07510ⓣ 201-523-5152
Henry Helstoskiⓣ 666 Paterson Ave., East Rutherford 07073ⓣ 201-939-9090
Peter W. Rodino, Jr.ⓣ Suite 1435A, 970 Broad St., Newark 07102ⓣ 201-645-3213
Joseph G. Minishⓣ 308 Main St., Orange 07050ⓣ 201-645-6363
Helen S. Meynerⓣ 32 Bridge St., Lambertville 08530ⓣ 609-397-1830
Dominick V. Danielsⓣ 895 Bergen Ave., Jersey City 07306ⓣ 201-659-7700
Edward J. Pattenⓣ Natl. Bank Bldg., Perth Amboy 08861ⓣ 201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent Fenwickⓣ 41 N. Bridge St., Somerville 08876ⓣ 201-722-8200
Edwin B. Forsytheⓣ 301 Mill St., Moorestown 08057ⓣ 609-235-6622
Matthew J. Rinaldoⓣ 1961 Morris Ave., Union 07083ⓣ 201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives		
(Democrats)		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
	(Republicans)	
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
Name ⓣ Address ⓣ Area Code ⓣ Phone
Holmdel ⓣ Holmdel, N. J. 07733 ⓣ 201 ⓣ 949-3000
Murray Hill ⓣ Murray Hill, N. J. 07974 ⓣ 201 ⓣ 582-6377
Whippany ⓣ Whippany, N. J. 07981 ⓣ 201 ⓣ 386-3000
Indian Hill ⓣ Naperville, Illinois 60540 ⓣ 312 ⓣ 690-2000
.TE
```

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb s s s
c|c|c s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places
_
Name Ⓣ Description Ⓣ Practical Information
_
T{
American Museum of Natural History
T}ⓉT{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors.  There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}ⓉHoursⓉ10-5, ex. Sun 11-5, Wed. to 9
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 79th St.
T}
\^Ⓣ\^ⓉAdmissionⓉDonation: $1.00 asked
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-4225
_
Bronx ZooⓉT{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}ⓉHoursⓉT{
10-4:30 winter, to 5:00 summer
T}
\^Ⓣ\^ⓉLocationⓉT{
185th St. & Southern Blvd, the Bronx.
T}
\^Ⓣ\^ⓉAdmissionⓉ$1.00, but Tu,We,Th free
\^Ⓣ\^ⓉSubwayⓉ2, 5 to East Tremont Ave.
\^Ⓣ\^ⓉTelephoneⓉ212-933-1759
_
Brooklyn MuseumⓉT{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}ⓉHoursⓉWed-Sat, 10-5, Sun 12-5
\^Ⓣ\^ⓉLocationⓉT{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉ2,3 to Eastern Parkway.
\^Ⓣ\^ⓉTelephoneⓉ212-638-5000
_
T{
New-York Historical Society
T}ⓉT{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}ⓉHoursⓉT{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 77th St.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

Acknowledgments.

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual,* Computing Science Technical Report No. 54, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," *Comm. ACM.* **17**, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. ACM.* **18**, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX,* UNIX Programmer's Manual, Volume 2.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX, Proc. AFIPS NCC*, vol. 46, pp. 879-888 (1977).

[6] J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 177-181 (October, 1976).

List of Tbl Command Characters and Words

<table>
<tr><th>Command</th><th>Meaning</th><th>Section</th></tr>
<tr><td>a A</td><td>Alphabetic subcolumn</td><td>2</td></tr>
<tr><td>allbox</td><td>Draw box around all items</td><td>1</td></tr>
<tr><td>b B</td><td>Boldface item</td><td>2</td></tr>
<tr><td>box</td><td>Draw box around table</td><td>1</td></tr>
<tr><td>c C</td><td>Centered column</td><td>2</td></tr>
<tr><td>center</td><td>Center table in page</td><td>1</td></tr>
<tr><td>doublebox</td><td>Doubled box around table</td><td>1</td></tr>
<tr><td>e E</td><td>Equal width columns</td><td>2</td></tr>
<tr><td>expand</td><td>Make table full line width</td><td>1</td></tr>
<tr><td>f F</td><td>Font change</td><td>2</td></tr>
<tr><td>i I</td><td>Italic item</td><td>2</td></tr>
<tr><td>l L</td><td>Left adjusted column</td><td>2</td></tr>
<tr><td>n N</td><td>Numerical column</td><td>2</td></tr>
<tr><td>nnn</td><td>Column separation</td><td>2</td></tr>
<tr><td>p P</td><td>Point size change</td><td>2</td></tr>
<tr><td>r R</td><td>Right adjusted column</td><td>2</td></tr>
<tr><td>s S</td><td>Spanned item</td><td>2</td></tr>
<tr><td>t T</td><td>Vertical spanning at top</td><td>2</td></tr>
<tr><td>tab (x)</td><td>Change data separator character</td><td>1</td></tr>
<tr><td>T{ T}</td><td>Text block</td><td>3</td></tr>
<tr><td>v V</td><td>Vertical spacing change</td><td>2</td></tr>
<tr><td>w W</td><td>Minimum width value</td><td>2</td></tr>
<tr><td>.xx</td><td>Included troff command</td><td>3</td></tr>
<tr><td>|</td><td>Vertical line</td><td>2</td></tr>
<tr><td>||</td><td>Double vertical line</td><td>2</td></tr>
<tr><td>^</td><td>Vertical span</td><td>2</td></tr>
<tr><td>\^</td><td>Vertical span</td><td>3</td></tr>
<tr><td>=</td><td>Double horizontal line</td><td>2,3</td></tr>
<tr><td>_</td><td>Horizontal line</td><td>2,3</td></tr>
<tr><td>_</td><td>Short horizontal line</td><td>3</td></tr>
<tr><td>\Rx</td><td>Repeat character</td><td>3</td></tr>
</table>

A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty, copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cdots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}\, e^{mx} - \sqrt{b}}{\sqrt{a}\, e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows

what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language," by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a,b]$. Nor should it assume that that $\sqrt{a+b}$ can be replaced by $(a+b)^{1/2}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, "standard" things should happen automatically. Someone who types "x=y+z+1" should get "$x=y+z+1$". Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity ("it should work everywhere"), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of

characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing x=y+z+1 should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

```
x    =    y
   + z + 1
```

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi\int\sin(\omega t)dt$$

we write

```
f(t) = 2 pi int sin ( omega t )dt
```

Here spaces are *necessary* in the input to indicate that *sin, pi, int,* and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over:*

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup:*

$$x^2+y^2=z^2$$

is produced by

```
x sup 2 + y sup 2 = z sup 2
```

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z}. The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of ${x_i}^2$.

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2}=\frac{x^2}{a^2}+\frac{y^2}{b^2}$$

is produced by

```
{partial sup 2 f} over {partial x sup 2} =
x sup 2 over a sup 2 + y sup 2 over b sup 2
```

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over,* so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of

the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqrt* operator for making square roots of the appropriate size: "sqrt a+b" produces $\sqrt{a+b}$, and

```
x = {-b +- sqrt{b sup 2 -4ac}} over 2a
```

is

$$x=\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqrt* is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to*. To get

$$\sum_{i=0}^{\infty} x_i \rightarrow 0$$

we need only type

```
sum from i=0 to inf x sub i -> 0
```

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

```
lim from {x -> pi /2} ( tan~x) = inf
```

is

$$\lim_{x \rightarrow \pi/2} (\tan x) = \infty$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right:*

```
left [ x+y over 2a right ]~=~1
```

makes

$$\left[\frac{x+y}{2a} \right] = 1$$

A *left* need not have a corresponding *right*, as we shall see in the next example. Any characters may follow *left* and *right*, but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles*, which make vertical piles of objects. For example, to get

$$sign\,(x) \equiv \left\{ \begin{array}{rl} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{array} \right.$$

we can type

```
sign (x) ~==~ left {
  rpile {1 above 0 above -1}
  ~~lpile {if above if above if}
  ~~lpile {x>0 above x=0 above x<0}
```

The construction "left {" makes a left brace big enough to enclose the "rpile {...}", which is a right-justified pile of "above ... above ...". "lpile" makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: "size 12 bold {A~x~=~y}" will produce **A x = y**. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

```
lim~ roman "sup" ~x sub n = 0
```

to ensure that the supremum doesn't become a superscript:

$$\lim \mathrm{sup}\ x_n = 0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\underset{.}{x} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z+Z}$$

is made by typing

```
x dot under + x hat + y tilde
 + X hat + Y dotdot = z+Z bar
```

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

```
define name "..."
```

at any time in the document; henceforth, any occurrence of the token "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own

specifications, for it is quite possible to redefine keywords like *sup* or *over*. Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ".ce" means "center the next output line"), EQN uses the sequence ".EQ" to mark the beginning of an equation and ".EN" to mark the end. The ".EQ" and ".EN" are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ".EQ" and ".EN" are simply ignored by TROFF, so by default equations are printed in-line.

".EQ" and ".EN" can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ".EQ" and ".EN" around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to "#", the input:

Let #x sub i#, #y# and #alpha# be positive

produces:

Let x_i, y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file "f", one issues the command:

```
eqn f | troff
```

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of "boxes," pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [ ] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn  : box | eqn box

box  : text
     | { eqn }
     | box OVER box
     | SQRT box
     | box SUB box | box SUP box
     | [ L | C | R ]PILE { list }
     | LEFT text eqn [ RIGHT text ]
     | box [ FROM box ] [ TO box ]
     | SIZE text box
     | [ROMAN | BOLD | ITALIC] box
     | box [HAT | BAR | DOT | DOTDOT | TILDE]
     | DEFINE text text

list  : eqn | list ABOVE eqn

text  : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn : box | eqn box
box : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

a over b over c ?

Is it

{a over b} over c

or is it

a over {b over c} ?

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right,

because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

    a sup 2 over b

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

    text   : TEXT

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

    box   : box OVER box

is:

    Width of output box =
      slightly more than largest input width
    Height of output box =
      slightly more than sum of input heights
    Base of output box =
      slightly more than height of bottom input box
    String describing output box =
      move down;
      move right enough to center bottom box;
      draw bottom box (i.e., copy string for bottom box);
      move up; move left enough to center top box;
      draw top box (i.e., copy string for top box);
      move down and left; draw line full width;
      return to proper base line.

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$${}^{l}_{m}\mathbf{T}^{kj}_{ni}$$

Grammatically, this is easy: it is sufficient to add a production like

    box   : TENSOR { list }

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being "easy to use," and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the

time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

(1) It's easy to write, although I make the following mistakes...

(2) How do I do...?

(3) It botches the following things.... Why don't you fix them?

(4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

```
f(x sub i)
```

which produces

$$f(x_{i)}$$

instead of

$$f(x_i)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
  {a sub 1 + b sub 2 over
    {a sub 2 + b sub 3 over
      {a sub 3 + ... }}}
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx} ~=~
left { lpile {
    1 over {2 mab} ~log~
        {sa emx - sb} over {sa emx + sb}
  above
    1 over mab ~ tanh sup -1 ( sa over sb emx )
  above
    -1 over mab ~ coth sup -1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler

and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

[1] *A Manual of Style*, 12th Edition. University of Chicago Press, 1969. p 295.

[2] *Model C/A/T Phototypesetter*. Graphic Systems, Inc., Hudson, N. H.

[3] Ritchie, D. M., and Thompson, K. L., "The UNIX time-sharing system." *Comm. ACM 17*, 7 (July 1974), 365-375.

[4] Oşsanna, J. F., TROFF User's Manual. Bell Laboratories Computing Science Technical Report 54, 1977.

[5] Aho, A. V., and Johnson, S. C., "LR Parsing." *Comp. Surv. 6*, 2 (June 1974), 99-124.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice-Hall, Inc., 1978.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k}\right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots\right)\left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots\right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \, \frac{S_2^{k_2}}{2^{k_2} k_2!} \, \cdots \, \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF[1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most comn.on way is to use the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows you to center, indent, left-justify and number equations.

With the '−ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (3.1a)$$

There is also a shorthand notation so in-line expressions like π_i^2 can be entered without .EQ and .EN. We will talk about it in section 19.

3. Input spaces

Spaces and newlines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between .EQ and .EN,

```
x=y+z
```

and

```
x = y + z
```

and

```
x   =   y
   + z
```

and so on all produce the same output

$$x=y+z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output*, use a tilde "~" for each space you want:

```
x~=~y~+~z
```

gives

$$x \ = \ y \ + \ z$$

You can also use a circumflex "^", which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
x=2 pi int sin ( omega t)dt
```

produces

$$x=2\pi\int\sin(\omega t)\,dt$$

Here the spaces in the input are **necessary** to tell EQN that *int, pi, sin* and *omega* are separate entities that should get special treatment. The *sin*, digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type *f(pi)* without leaving spaces on both sides of the *pi*. As a result, EQN does not recognize *pi* as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like \(*bs* for the Bell System sign ⓑ.

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

is much the same as the last example, except that the tildes not only separate the magic words like *sin, omega,* and so on, but also add extra spaces, one space per tilde:

$$x \ = \ 2 \ \pi \ \int \ \sin \ (\ \omega \ t \ ) \ dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup*.

```
x sup 2 + y sub k
```

gives

$$x^2+y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you *xsub*2 instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

x sub i sub 1

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first:*

x sub i sup 2

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means x^{y_z}, not ${x^y}_z$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

e sup {i omega t}

is

$$e^{i\omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

x sub {i sub 1} sup 2

is

$$x_{i_1}^2$$

with braces, but

x sub i sub 1 sup 2

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

e sup {i pi sup {rho +1}}

is

The general rule is that anywhere you could use some single thing like *x*, you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over:*

a+b over 2c =1

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

{alpha + beta} over {sin (x)}

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over*, so

−b sup 2 over pi

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt:*

sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

```
sqrt {a sup 2 over b sub 2}
```

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

$$(a^2/b_2)^{1/2}$$

which is

```
(a sup 2 /b sub 2 ) sup half
```

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

```
sum from i=0 to {i= inf} x sup i
```

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

```
int  prod  union  inter
```

become, respectively,

$$\int \quad \prod \quad \bigcup \quad \bigcap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

```
lim from {n -> inf} x sub n =0
```

is

$$\lim_{n \to \infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman, italic, bold* and *fat.* Like *sub* and *sup,* size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

```
bold x y
```

is

$$\mathbf{x} y$$

and

```
size 14 bold x = y +
  size 14 {alpha + beta}
```

gives

$$\mathbf{x} = y + \alpha + \beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

```
size 12 { ... }
```

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size −3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is $\boldsymbol{x_i}$.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which

thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or −.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

```
italic "sin(x)" + sin (x)
```

is

sin(x)+sin(*x*)

Quotes are also used to get braces and other EQN keywords printed:

```
"{ size alpha }"
```

is

{ size alpha }

and

```
roman "{ size alpha }"
```

is

{ size alpha }

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ^{2}He, you can't just type *sup 2 roman He* because a *sup* has to be a superscript *on* something. Thus you must say

```
"" sup 2 roman He
```

To get a literal quote use "\"". TROFF characters like \(*bs* can appear unquoted, but more complicated things like horizontal and vertical motions with *h* and *v* should always be quoted. (If you've never heard of *h* and *v*, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. Thus, for example, you can say

---

‡Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set by the command *gsize −2*.

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$

$$x=1$$

For reasons too complicated to talk about, when you use EQN and '−ms', use either .EQ I or .EQ L. mark and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

```
x mark =1
...
x+y lineup =z
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

16. Big Brackets, Etc.

To get big brackets [ ], braces { }, parentheses ( ), and bars || around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
~=~ left ( c over d right )
+ left [ e right ]
```

is

$$\left\{ \frac{a}{b}+1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a "left nothing". This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
```

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l-*, *r-* and *cpiles* than it is for ordinary piles.

```
roman sign (x)~=~
left {
  lpile {1 above 0 above -1}
  ~~ lpile
  {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\mathrm{sign}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{cases}$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it.* The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like x italic. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '−ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

```
Let $alpha sub i$ be the primary
variable, and let $beta$ be zero.
Then we can show that $x sub 1$ is
$>=0$.
```

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

This makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. You can use any character

instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi ' x sub i '
define xi1 ' xi sub 1 '
.EN
```

don't define something in terms of itself. A favorite error is to say

```
define X ' roman X '
```

This is a guaranteed disaster, since X *is* now defined in terms of itself. If you say

```
define X ' roman "X" '
```

however, the quotes protect the second X, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

```
define / ' over '
```

or redefine *over* as / with

```
define over ' / '
```

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~  prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup =  sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ,..., k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

These character sequences are recognized and translated as shown.

Sequence	Result
>=	$\geq$
<=	$\leq$
==	$\equiv$
!=	$\neq$
+-	$\pm$
->	$\rightarrow$
<-	$\leftarrow$
<<	$\ll$
>>	$\gg$
inf	∞
partial	∂
half	$\frac{1}{2}$
prime	$'$
approx	$\approx$
nothing	
cdot	$\cdot$
times	$\times$
del	∇
grad	∇
...	$\cdots$
,...,	$,\ldots,$
sum	$\sum$
int	$\int$
prod	$\prod$
union	$\bigcup$
inter	$\bigcap$

To obtain Greek letters, simply spell them out in whatever case you want:

Name	Letter	Name	Letter
DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	o
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

Word	Section	Word	Section
above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~, ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

syntax error between lines x and y, file z

where x and y are approximately the lines between which the trouble occurred, and z is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

eqn files >/dev/null

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message "word overflow", you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message "line overflow" indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate .EQ ... .EN sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

eqn files | troff

If there are any TROFF options, they go after the TROFF part of the command. For example,

eqn files | troff −ms

To run the same document on the GCOS typesetter, use

eqn files | troff −g (other options) | gcat

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

neqn files | nroff

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

neqn files | nroff −T*x*

where *x* is the terminal type you are using, such as *300* or *300S*.

EQN and NEQN can be used with the TBL program[2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

tbl files | eqn | troff
tbl files | neqn | nroff

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

[1] J. F. Ossanna, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories, 1976.

[3] M. E. Lesk, "TBL — A Program for Setting Tables", Bell Laboratories Computing Science Technical Report #49, 1976.

DOCUMENTS FOR USE WITH THE UNIX TIME-SHARING SYSTEM

Hybrid PWB - V7

The enclosed UNIX documentation is supplied
in accordance with the Software Agreement
you have with the Western Electric Company.

Western Electric

Guilford Center
P. O. Box [illegible]
Greensboro, NC 274[illegible]
919 697 [illegible]

Patent Licensing

OCT 03 1980

TECHNISCHE HOGESCHOOL DELFT
Department of Mathematics
Juianalaan 132
2628 BL Delft
The Netherlands

Attn: Mr. P. J. van der Hoff

Gentlemen:

Re: May 1, 1979 Software Agreement Between Us
Relating to PWB/UNIX* Time Sharing Operating
System

In response to the August 22, 1980 request from Mr. van der Hoff, your institution may use the licensed software pursuant to the referenced agreement on the following additional CPU's:

LSI-11, Serial No. WM 1720
PDP 11/60, Serial No. AG 00073
Technische Hogeschool Delft
Department of Mathematics - Comp. Rm. 0.101
Julianalaan 132
2628 BL Delft
The Netherlands

Yours truly,

O. L. Wilson

O. L. WILSON
Patent Licensing Manager

*UNIX is a trademark of Bell Laboratories.

Copyright 1979, Bell Telephone Laboratories, Incorporated. Holders of a UNIX™ software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

CONTENTS

1. Programming in C - A Tutorial
2. The C Programming Language - Reference Manual
3. UNIX Programming - Second Edition
4. A Tutorial Introduction to ADB
5. Lex - A Lexical Analyzer Generator
6. Make - A Program for Maintaining Computer Programs
7. LINT - a C Program Checker
8. Yacc - Yet Another Compiler-Compiler

Programming in C — A Tutorial

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

1. Introduction

C is a computer language available on the GCOS and UNIX operating systems at Murray Hill and (in preliminary form) on OS/360 at Holmdel. C lets you write your programs clearly and simply — it has decent control flow facilities so your code can be read straight down the page, without labels or GOTO's; it lets you write code that is compact without being too cryptic; it encourages modularity and good program organization; and it provides good data-structuring facilities.

This memorandum is a tutorial to make learning C as painless as possible. The first part concentrates on the central features of C; the second part discusses those parts of the language which are useful (usually for getting more efficient and smaller code) but which are not necessary for the new user. This is *not* a reference manual. Details and special cases will be skipped ruthlessly, and no attempt will be made to cover every language feature. The order of presentation is hopefully pedagogical instead of logical. Users who would like the full story should consult the *C Reference Manual* by D. M. Ritchie [1], which should be read for details anyway. Runtime support is described in [2] and [3]; you will have to read one of these to learn how to compile and run a C program.

We will assume that you are familiar with the mysteries of creating files, text editing, and the like in the operating system you run on, and that you have programmed in some language before.

2. A Simple C Program

```
main( ) {
        printf("hello, world");
}
```

A C program consists of one or more *functions*, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. **main** is such a function, and in fact all C programs must have a **main**. Execution of the program begins at the first statement of **main**. **main** will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here **main** is a function of no arguments, indicated by ( ). The {} enclose the statements of the function. Individual statements end with a semicolon but are otherwise free-format.

printf is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A function is invoked by naming it, followed by a list of arguments in parentheses. There is no CALL statement as in Fortran or PL/I.

3. A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {
        int a, b, c, sum;
        a = 1;  b = 2;  c = 3;
        sum = a + b + c;
        printf("sum is %d", sum);
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has four fundamental *types* of variables:

- **int** integer (PDP-11: 16 bits; H6070: 36 bits; IBM360: 32 bits)
- **char** one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
- **float** single-precision floating point
- **double** double-precision floating point

There are also *arrays* and *structures* of these basic types, *pointers* to them and *functions* that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

declares **a**, **b**, **c**, and **sum** to be integers.

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and _, and start with a non-digit. Stylistically, it's much better to use only a single case and give functions and external variables names that are unique in the first six characters. (Function and external variable names are used by various assemblers, some of which are limited in the size and case of identifiers they can handle.) Furthermore, keywords and library functions may only be recognized in one case.

4. Constants

We have already seen decimal integer constants in the previous example — 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus 0777 is an octal constant, with decimal value 511.

A "character" is one byte (an inherently machine-dependent concept). Most often this is expressed as a *character constant*, which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in **flags** below:

```
char quest, newline, flags;
quest = '?';
newline = '\n';
flags = 077;
```

The sequence '\n' is C notation for "newline character", which, when printed, skips the terminal to the beginning of the next line. Notice that '\n' represents only a single character. There are several other "escapes" like '\n' for representing hard-to-get or invisible characters, such as '\t' for tab, '\b' for backspace, '\0' for end of file, and '\\' for the backslash itself.

float and **double** constants are discussed in section 26.

5. Simple I/O — getchar, putchar, printf

```
main( ) {
        char c;
        c = getchar( );
        putchar(c);
}
```

getchar and **putchar** are the basic I/O library functions in C. **getchar** fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by '\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

putchar puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

printf is a more complicated function for producing formatted output. We will talk about only the simplest use of it. Basically, **printf** uses its first argument as formatting information, and any successive arguments as variables to be output. Thus

```
printf ("hello, world\n");
```

is the simplest use — the string "hello, world\n" is printed out. No formatting information, no variables, so the string is dumped out verbatim. The newline is necessary to put this out on a line by itself. (The construction

```
"hello, world\n"
```

is really an array of **chars**. More about this shortly.)

More complicated, if **sum** is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of **printf**, the characters "%d" signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are "%c" to print out a single character, "%s" to print out an entire string, and "%o" to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
```

```
printf (" %s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal? Right! 511 decimal is 777 octal
```

Notice that there is no newline at the end of the first output line. Successive calls to **printf** (and/or **putchar**, for that matter) simply put out characters. No newlines are printed unless you ask for them. Similarly, on input, characters are read one at a time as you ask for them. Each line is generally terminated by a newline (\n), but there is otherwise no concept of record.

6. If; relational operators; compound statements

The basic conditional-testing statement in C is the **if** statement:

```
c = getchar( );
if( c == '?' )
        printf("why did you type a question mark?\n");
```

The simplest form of **if** is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional **else** clause, to be described soon.

The character sequence '==' is one of the relational operators in C; here is the complete set:

```
==    equal to (.EQ. to Fortraners)
!=    not equal to
>     greater than
<     less than
>=    greater than or equal to
<=    less than or equal to
```

The value of "**expression relation expression**" is 1 if the relation is true, and 0 if false. Don't forget that the equality test is '=='; a single '=' causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators '&&' (AND), '||' (OR), and '!' (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' ) ...
```

C guarantees that '&&' and '||' are evaluated left to right — we shall soon see cases where this matters.

One of the nice things about C is that the **statement** part of an **if** can be made arbitrarily complicated by enclosing a set of statements in {}. As a simple example, suppose we want to ensure that **a** is bigger than b, as part of a sort routine. The interchange of **a** and **b** takes three statements in C, grouped together by {}:

```
if (a < b) {
        t = a;
        a = b;
        b = t;
}
```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in {}. There is no semicolon after the } of a compound statement, but there *is* a semicolon after the last non-compound statement inside the {}.

The ability to replace single statements by complex ones at will is one feature that makes C much more pleasant to use than Fortran. Logic (like the exchange in the previous example) which would require several GOTO's and labels in Fortran can and should be done in C without any, using compound statements.

7. While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the **while** statement. Here's a program that copies its input to its output a character at a time. Remember that '\0' marks the end of file.

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                putchar(c);
}
```

The **while** statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

```
(a) evaluate the expression
(b) if its value is true (i.e., not zero)
            do the statement, and go back to (a)
```

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the **if** statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to c, and then tests if it's a '\0''. If it is not a '\0', the statement part of the **while** is executed, printing the character. The **while** then repeats. When the input character is finally a '\0', the **while** terminates, and so does **main.**

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, re-write the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

c would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator '=' is evaluated after the relational operator '!='. When in doubt, or even if not, parenthesize.

Since **putchar(c)** returns **c** as its function value, we could also copy the input to the output by nesting the calls to **getchar** and **putchar**:

```
main( ) {
        while( putchar(getchar( )) != '\0' ) ;
}
```

What statement is being repeated? None, or technically, the *null* statement, because all the work is really done within the test part of the **while**. This version is slightly different from the previous one, because the final '\0' is copied to the output before we decide to stop.

8. Arithmetic

The arithmetic operators are the usual '+', '−', '*', and '/' (truncating integer division if the operands are both **int**), and the remainder or mod operator '%':

```
x = a%b;
```

sets *x* to the remainder after **a** is divided by **b** (i.e., **a mod b**). The results are machine dependent unless **a** and **b** are both positive.

In arithmetic, **char** variables can usually be treated like **int** variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' - 'a';
```

converts a single lower case ascii character stored in **c** to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all **chars** are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension — if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                if( 'A'<=c && c<='Z' )
                        putchar(c+'a'-'A');
                else
                        putchar(c);
}
```

Characters have different sizes on different machines. Further, this code won't work on an IBM machine, because the letters in the ebcdic alphabet are not contiguous.

9. Else Clause; Conditional Expressions

We just used an **else** after an **if**. The most general form of **if** is

```
if (expression) statement1 else statement2
```

the **else** part is optional, but often useful. The canonical example sets **x** to the minimum of **a** and **b**:

```
if (a < b)
        x = a;
else
        x = b;
```

Observe that there s a semicolon after **x=a**.

C provides an alternate form of conditional which is often more concise. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a<b ? a : b;
```

is a if a is less than b; it is b otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

means "evaluate expr1. If it is not zero, the value of the whole thing is expr2; otherwise the value is expr3."

To set x to the minimum of a and b, then:

```
x = (a<b ? a : b);
```

The parentheses aren't necessary because '?:' is evaluated before '=', but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
	putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(...)
	{...}
else if(...)
	{...}
else if(...)
	{...}
else
	{...}
```

The conditions are tested in order, and exactly one block is executed — either the first one whose if is satisfied, or the one for the last else. When this block is finished, the next statement executed is the one after the last else. If no action is to be taken for the "default" case, omit the last else.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
	int let, dig, other, c;
	let = dig = other = 0;
	while( (c=getchar( )) != '\0' )
		if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )  ++let;
		else if( '0'<=c && c<='9' )  ++dig;
		else  ++other;
	printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

The '++' operator means "increment by 1"; we will get to it in the next section.

10. Increment and Decrement Operators

In addition to the usual '−', C also has two other interesting unary operators, '++' (increment) and '−−' (decrement). Suppose we want to count the lines in a file.

```
main( ) {
	int c,n;
	n = 0;
```

```
        while( (c=getchar( )) != '\0' )
                if( c == '\n' )
                        ++n;
        printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. '++' and '−−' can be applied only to int's and char's (and pointers which we haven't got to yet).

The unusual feature of '++' and '−−' is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets x to to 5, and *then* increments k to 6. The incrementing effect of ++k and k++ is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

11. Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean *subscripting*; parentheses are used only for function references. Array indexes begin at *zero*, so the elements of x are

```
x[0], x[1], x[2], ..., x[9]
```

If an array has n elements, the largest subscript is n−1.

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; name has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
        int n, c;
        char line[100];
        n = 0;
        while( (c=getchar( )) != '\n' ) {
                if( n < 100 )
                        line[n] = c;
                n++;
        }
        printf("length = %d\n", n);
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
        int n, c; char line[100];
        n = 0;
        while( (c=getchar( )) != '\0' )
                if( c == '\n' ) {
                        printf("%d\n", n);
                        n = 0;
                }
                else {
                        if( n < 100 ) line[n] = c;
                        n++;
                }
}
```

12. Character Arrays; Strings

Text is usually kept as an array of characters, as we did with **line[ ]** in the example above. By convention in C, the last character in a character array should be a '\0' because most programs that manipulate character arrays expect it. For example, **printf** uses the '\0' to detect the end of a character array when printing it out with a '%s'.

We can copy a character array **s** into another **t** like this:

```
        i = 0;
        while( (t[i]=s[i]) != '\0' )
                i++;
```

Most of the time we have to put in our own '\0' at the end of a string; if we want to print the line with **printf**, it's necessary. This code prints the character count before the line:

```
main( ) {
        int n;
        char line[100];
        n = 0;
        while( (line[n++]=getchar( )) != '\n' );
        line[n] = '\0';
        printf("%d:\t%s", n, line);
}
```

Here we increment **n** in the subscript itself, but only after the previous value has been used. The character is read, placed in **line[n]**, and only then **n** is incremented.

There is one place and one place only where C puts in the '\0' at the end of a character array for you, and that is in the construction

"stuff between double quotes"

The compiler puts a '\0' at the end automatically. Text enclosed in double quotes is called a *string*; its properties are precisely those of an (initialized) array of characters.

13. For Statement

The **for** statement is a somewhat generalized **while** that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the **for** is

```
for( initialization; expression; increment )
        statement
```

The meaning is exactly

```
initialization;
while( expression ) {
        statement
        increment;
}
```

Thus, the following code does the same array copy as the example in the previous section:

```
for( i=0; (t[i]=s[i]) != '\0'; i++ );
```

This slightly more ornate example adds up the elements of an array:

```
sum = 0;
for( i=0; i<n; i++)
        sum = sum + array[i];
```

In the **for** statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the **while**: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the **while**, the **for** loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
for( ; ; ) ...
```

and

```
while( 1 ) ...
```

are both infinite loops.

You might ask why we use a **for** since it's so much like a **while**. (You might also ask why we use a **while** because...) The **for** is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

```
for( i=0; i<n; i++ )
        for( j=0; j<m; j++ )
                array[i][j] = 0;
```

14. Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value>127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
        int hist[129];          /* 128 legal chars + 1 illegal group */
        ...
        count(hist, 128);       /* count the letters into hist */
        printf( ... );          /* comments look like this; use them */
        ...                 /* anywhere blanks, tabs or newlines could appear */
}

count(buf, size)
   int size, buf[ ]; {
        int i, c;
        for( i=0; i<=size; i++ )
                buf[i] = 0;                     /* set buf to zero */
        while( (c=getchar( )) != '\0' ) {       /* read til eof */
                if( c > size || c < 0 )
                        c = size;               /* fix illegal input */
                buf[c]++;
        }
        return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to *define* one. Since count has two arguments, we need to declare them, as shown, giving their types, and in the case of buf, the fact that it is an array. The declarations of arguments go *between* the argument list and the opening '{'. There is no need to specify the size of the array buf, for it is defined outside of count.

The return statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted count to return a value, say the number of characters read? The return statement allows for this too:

```
int i, c, nchar;
nchar = 0;
...
while( (c=getchar( )) != '\0' ) {
        if( c > size || c < 0 )
                c = size;
        buf[c]++;
        nchar++;
}
return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
   int a, b; {
        return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcopy(s1, s2)          /* copies s1 to s2 */
   char s1[ ], s2[ ]; {
        int i;
        for( i = 0; (s2[i] = s1[i]) != '\0'; i++ );
}
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the **for**. Again, the declarations of the arguments **s1** and **s2** omit the sizes, because they don't matter to **strcopy**. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by "call by value", which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

15. Local and External Variables

If we say

```
f( ) {
        int x;
        ...
}
g( ) {
        int x;
        ...
}
```

each **x** is *local* to its own routine — the **x** in **f** is unrelated to the **x** in **g**. (Local variables are also called "automatic".) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a **static** storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, *external variables* are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to *define* them external to all functions, and, wherever we want to use them, make a *declaration*.

```
main( ) {
        extern int nchar, hist[ ];
        ...
        count( );
        ...
}
```

```
count( ) {
        extern int nchar, hist[ ];
        int i, c;
        ...
}
int     hist[129];      /* space for histogram */
int     nchar;          /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an **extern** declaration for it. The declaration is the same as others, except for the added keyword **extern**. Furthermore, there must somewhere be a ***definition*** of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

```
int     nchar 0;
char    flag    'f';
  etc.
```

This is discussed further in a later section.

This ends our discussion of what might be called the central core of C. You now have enough to write quite substantial C programs, and it would probably be a good idea if you paused long enough to do so. The rest of this tutorial will describe some more ornate constructions, useful but not essential.

16. Pointers

A *pointer* in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator '&' is used to produce the address of an object, if it has one. Thus

```
int a, b;
b = &a;
```

puts the address of **a** into **b**. We can't do much with it except print it or pass it to some other routine, because we haven't given **b** the right kind of declaration. But if we declare that **b** is indeed a *pointer* to an integer, we're in good shape:

```
int a, *b, c;
b = &a;
c = *b;
```

b contains the address of **a** and '**c = *b**' means to use the value in **b** as an address, i.e., as a pointer. The effect is that we get back the contents of **a**, albeit rather indirectly. (It's always the case that '***&x**' is the same as **x** if **x** has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y;
char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make **y** point to an element of **x** by either of

point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent.

Now '*y' gives x[0]. More importantly,

```
*(y+1) gives x[1]
*(y+i) gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

leaves y pointing at x[1].

Let's use pointers in a function **length** that computes how long a character array is. Remember that by convention all character arrays are terminated with a '\0'. (And if they aren't, this program will blow up inevitably.) The old way:

```
length(s)
   char s[ ]; {
        int n;
        for( n=0; s[n] != '\0'; )
                n++;
        return(n);
}
```

Rewriting with pointers gives

```
length(s)
   char *s; {
        int n;
        for( n=0; *s != '\0'; s++ )
                n++;
        return(n);
}
```

You can now see why we have to say what kind of thing **s** points to — if we're to increment it with **s++** we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
        for( n=0; *s++ != '\0'; n++ );
```

The '*s' returns a character; the '++' increments the pointer so we'll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But '*s++' is an idiom so common that you have to know it.

Going a step further, here's our function **strcopy** that copies a character array **s** to another **t**.

```
strcopy(s,t)
   char *s, *t; {
        while(*t++ = *s++);
}
```

We have omitted the test against '\0', because '\0' is identically zero; you will often see the code this way. (You *must* have a space after the '=': see section 25.)

For arguments to a function, and there only, the declarations

```
char s[ ];
char *s;
```

are equivalent — a pointer to a type, or an array of unspecified size of that type, are the same thing.

If this all seems mysterious, copy these forms until they become second nature. You don't often need anything more complicated.

17. Function Arguments

Look back at the function strcopy in the previous section. We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don't lose the original strings in the function that called strcopy?

As we said before, C is a "call by value" language: when you make a function call like f(x), the *value* of x is passed, not its address. So there's no way to *alter* x from inside f. If x is an array (char x[10]) this isn't a problem, because x *is* an address anyway, and you're not trying to change it, just what it addresses. This is why strcopy works as it does. And it's convenient not to have to worry about making temporary copies of the input arguments.

But what if x is a scalar and you do want to change it? In that case, you have to pass the *address* of x to f, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)
   int *x, *y; {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

and to call flip, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

18. Multiple Levels of Pointers; Program Arguments

When a C program is called, the arguments on the command line are made available to the main program as an argument count argc and an array of character strings argv containing the arguments. Manipulating these arguments is one of the most common uses of multiple levels of pointers ("pointer to pointer to ..."). By convention, argc is greater than zero; the first argument (in argv[0]) is the command name itself.

Here is a program that simply echoes its arguments.

```
main(argc, argv)
   int argc;
   char **argv; {
        int i;
        for( i=1; i < argc; i++ )
                printf("%s ", argv[i]);
        putchar('\n');
}
```

Step by step: main is called with two arguments, the argument count and the array of arguments. argv is a pointer to an array, whose individual elements are pointers to arrays of char-

acters. The zeroth argument is the name of the command itself, so we start to print with the first argument, until we've printed them all. Each **argv[i]** is a character array, so we use a '**%s**' in the **printf**.

You will sometimes see the declaration of **argv** written as

```
char *argv[ ];
```

which is equivalent. But we can't use **char argv[ ][ ]**, because both dimensions are variable and there would be no way to figure out how big the array is.

Here's a bigger example using **argc** and **argv**. A common convention in C programs is that if the first argument is '−', it indicates a flag of some sort. For example, suppose we want a program to be callable as

```
prog -abc arg1 arg2 ...
```

where the '−' argument is optional; if it is present, it may be followed by any combination of a, b, and c.

```
main(argc, argv)
   int argc;
   char **argv; {
        ...
        aflag = bflag = cflag  = 0;
        if( argc > 1 && argv[1][0] == '-' ) {
                for( i=1; (c=argv[1][i]) != '\0'; i++ )
                        if( c=='a' )
                                aflag++;
                        else if( c=='b' )
                                bflag++;
                        else if( c=='c' )
                                cflag++;
                        else
                                printf("%c?\n", c);
                --argc;
                ++argv;
        }
        ...
```

There are several things worth noticing about this code. First, there is a real need for the left-to-right evaluation that && provides; we don't want to look at **argv[1]** unless we know it's there. Second, the statements

```
--argc;
++argv;
```

let us march along the argument list by one position, so we can skip over the flag argument as if it had never existed — the rest of the program is independent of whether or not there was a flag argument. This only works because **argv** is a pointer which can be incremented.

19. The Switch Statement; Break; Continue

The **switch** statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
if( c == 'a' ) ...
else if( c == 'b' ) ...
else if( c == 'c' ) ...
else ...
```

testing a value against a series of *constants*, the switch statement is often clearer and usually gives better code. Use it like this:

```
switch( c ) {

case 'a':
        aflag++;
        break;
case 'b':
        bflag++;
        break;
case 'c':
        cflag++;
        break;
default:
        printf("%c?\n", c);
        break;
}
```

The **case** statements label the various actions we want; **default** gets done if none of the other cases are satisfied. (A **default** is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The **break** statement in this example is new. It is there because the cases are just labels, and after you do one of them, you *fall through* to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower case letters in our flag field, so we could say

```
case 'a':  case 'A':    ...
case 'b':  case 'B':    ...
 etc.
```

But what if we just want to get out after doing **case 'a'** ? We could get out of a **case** of the **switch** with a label and a **goto**, but this is really ugly. The **break** statement lets us exit without either goto or label.

```
switch( c ) {

case 'a':
        aflag++;
        break;
case 'b':
        bflag++;
        break;
 ...
}
/* the break statements get us here directly */
```

The **break** statement also works in **for** and **while** statements — it causes an immediate exit from the loop.

The **continue** statement works *only* inside **for's** and **while's**; it causes the next iteration of the loop to be started. This means it goes to the increment part of the **for** and the test part of the **while**. We could have used a **continue** in our example to get on with the next iteration of the **for**, but it seems clearer to use **break** instead.

20. Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char   id[10];
int    line;
char   type;
int    usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
        char   id[10];
        int    line;
        char   type;
        int    usage;
} sym;
```

This defines sym to be a structure with the specified shape; **id**, **line**, **type** and **usage** are *members* of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
   etc.
```

Although the names of structure members never stand alone, they still have to be unique — there can't be another id or **usage** in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char   id[100][10];
int    line[100];
char   type[100];
int    usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifer is collected into one lump:

```
struct {
        char   id[10];
        int    line;
        char   type;
        int    usage;
} sym[100];
```

This makes sym an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++;      /* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
  etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
        if( sym[i].usage == 0 )
                printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function lookup(name) which will tell us if name already exists in sym, by giving its index, or that it doesn't, by returning a −1. We can't pass a structure to a function directly — we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int     nsym    0;      /* current length of symbol table */

struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];             /* symbol table */

main( ) {
        ...
        if( (index = lookup(newname)) >= 0 )
                sym[index].usage++;          /* already there ... */
        else
                install(newname, newline, newtype);
        ...
}

lookup(s)
   char *s; {
        int i;
        extern struct {
                char    id[10];
                int     line;
                char    type;
                int     usage;
        } sym[ ];

        for( i=0; i<nsym; i++ )
                if( compar(s, sym[i].id) > 0 )
                        return(i);
        return(-1);
}

compar(s1,s2)           /* return 1 if s1==s2, 0 otherwise */
   char *s1, *s2; {
        while( *s1++ == *s2 )
                if( *s2++ == '\0' )
                        return(1);
```

```
        return(0);
    }
```

The declaration of the structure in **lookup** isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```
    struct  symtag {
            char    id[10];
            int     line;
            char    type;
            int     usage;
    } sym[100], *psym;

            psym = &sym[0];         /* or psym = sym; */
```

This makes **psym** a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of **sym.**

Notice that we added something after the word **struct**: a "tag" called **symtag.** This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
    struct  symtag {
            ... structure definition
    };
```

which wouldn't have assigned any storage at all, and then said

```
    struct  symtag sym[100];
    struct  symtag *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
    struct  symtag sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
            ptr -> structure-member
```

The symbol '->' means we're pointing at a member of a structure; '->' is only used in that context. **ptr** is a pointer to the (base of) a structure that contains the structure member. The expression **ptr->structure-member** refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
    psym->type = 1;
    psym->id[0] = 'a';
```

and so on.

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
    struct { int x, *y; } *p;
    p->x++        increments x
    ++p->x        so does this!
    (++p)->x      increments p before getting x
    *p->y++       uses y as a pointer, then increments it
    *(p->y)++     so does this
    *(p++)->y     uses y as a pointer, then increments p
```

The way to remember these is that ->, . (dot), ( ) and [ ] bind very tightly. An expression in-

volving one of these is treated as a unit. **p->x**, **a[i]**, **y.x** and **f(b)** are names exactly as **abc** is.

If **p** is a pointer to a structure, any arithmetic on **p** takes into account the acutal size of the structure. For instance, p++ increments p by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members — because of alignments of different sized objects, there may be "holes" in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```
struct symtag {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];

main( ) {
        struct symtag *lookup( );
        struct symtag *psym;
        ...
        if( (psym = lookup(newname)) )      /* non-zero pointer */
                psym -> usage++;            /* means already there */
        else
                install(newname, newline, newtype);
        ...
}

struct symtag *lookup(s)
   char *s; {
        struct symtag *p;
        for( p=sym; p < &sym[nsym]; p++ )
                if( compar(s, p->id) > 0)
                        return(p);
        return(0);
}
```

The function **compar** doesn't change: 'p->id' refers to a string.

In **main** we test the pointer returned by **lookup** against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```
struct symtag *lookup( );
```

This brings us to an area that we will treat only hurriedly — the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an **int** has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see). Examples:

```
char f(a)
   int a; {
        ...
}
```

```
int *g( ) { ... }

struct symtag *lookup(s) char *s; { ... }
```

The function f returns a character, g returns a pointer to an integer, and lookup returns a pointer to a structure that looks like symtag. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in main above.

Notice the parallelism between the declarations

```
struct symtag *lookup( );
struct symtag *psym;
```

In effect, this says that lookup( ) and psym are both used the same way — as a pointer to a strcture — even though one is a variable and the other is a function.

21. Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int     x       0;      /* "0" could be any constant */
int     a       'a';
char    flag    0177;
int     *p      &y[1];  /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int     x[4]    {0,1,2,3};              /* makes x[i] = i */
int     y[ ]    {0,1,2,3};              /* makes y big enough for 4 values */
char    *msg    "syntax error\n";       /* braces unnecessary here */
char *keyword[ ]{
        "if",
        "else",
        "for",
        "while",
        "break",
        "continue",
        0
};
```

This last one is very useful — it makes keyword an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)             /* search for str in keyword[ ] */
  char *str; {
        int i,j,r;
        for( i=0; keyword[i] != 0; i++) {
                for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++ );
                if( r == str[j] )
                        return(i);
        }
        return(-1);
}
```

Sorry — neither local variables nor structures can be initialized.

22. Scope Rules: Who Knows About What

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words *declaration* and *definition* are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making **extern** declarations. If the definition of a variable appears *before* its use in some function, no **extern** declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }
int foo;
f2( ) { ... foo = 1; ... }
f3( ) { ... if ( foo ) ... }
```

no declaration of foo is needed in either f2 or or f3, because the external definition of **foo** appears before them. But if f1 wants to use foo, it has to contain the declaration

```
f1( ) {
        extern int foo;
        ...
}
```

This is true also of any function that exists on another file — if it wants **foo** it has to use an **extern** declaration for it. (If somewhere there is an **extern** declaration for something, there must also eventually be an external definition of it, or you'll get an "undefined symbol" message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int     foo     0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an **extern** declaration, outside of any function:

```
extern int     foo;
f1( ) { ... }
  etc.
```

The **#include** compiler control line, to be discussed shortly, lets you make a single copy of the external declarations for a program and then stick them into each of the source files making up the program.

23. #define, #include

C provides a very limited macro facility. You can say

```
#define     name          something
```

and thereafter anywhere "name" appears as a token, "something" will be substituted. This is

particularly useful in parametering the sizes of arrays:

```
#define      ARRAYSIZE   100
     int     arr[ARRAYSIZE];
     ...
     while( i++ < ARRAYSIZE )...
```

(now we can alter the entire program by changing only the **define**) or in setting up mysterious constants:

```
#define      SET         01
#define      INTERRUPT   02      /* interrupt bit */
#define      ENABLED     04
...
if( x & (SET | INTERRUPT | ENABLED) ) ...
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators '&' (AND) and '|' (OR) will be covered in the next section.) It's an excellent practice to write programs without any literal constants except in **#define** statements.

There are several warnings about **#define**. First, there's no semicolon at the end of a **#define**; all the text from the name to the end of the line (except for comments) is taken to be the "something". When it's put into the text, blanks are placed around it. Good style typically makes the name in the **#define** upper case — this makes parameters more visible. Definitions affect things only after they occur, and only within the file in which they occur. Defines can't be nested. Last, if there is a **#define** in a file, then the first character of the file *must* be a '#', to signal the preprocessor that definitions exist.

The other control word known to C is **#include**. To include one file in your source at compilation time, say

```
#include "filename"
```

This is useful for putting a lot of heavily used data definitions and **#define** statements at the beginning of a file to be compiled. As with **#define**, the first line of a file containing a **#include** has to begin with a '#'. And **#include** can't be nested — an included file can't contain another **#include**.

24. Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of x and 0177, effectively retaining only the last seven bits of x. Other operators are

|	inclusive OR
^	(circumflex) exclusive OR
~	(tilde) 1's complement
!	logical NOT
<<	left shift (as in x<<2)
>>	right shift (arithmetic on PDP-11; logical on H6070, IBM360)

25. Assignment Operators

An unusual feature of C is that the normal binary operators like '+', '−', etc. can be combined with the assignment operator '=' to form new assignment operators. For example,

```
x =- 10;
```

uses the assignment operator '=−' to decrement x by 10, and

```
x =& 0177
```

forms the AND of x and 0177. This convention is a useful notational shortcut, particularly if x is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
	sum =+ array[i];
```

But the spaces around the operator are critical! For instance,

```
x = -10;
```

sets x to −10, while

```
x =- 10;
```

subtracts 10 from x. When no space is present,

```
x=-10;
```

also decreases x by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means "shift x left y places, then OR with z, and store in x." But

```
x =<< y | z;
```

means "shift x left by y|z places", which is rather different.

26. Floating Point

We've skipped over floating point so far, and the treatment here will be hasty. C has single and double precision numbers (where the precision depends on the machine at hand). For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
	sum =+ y[i];
avg = sum/n;
```

forms the sum and average of the array y.

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands **int** or **char**, the arithmetic done is integer, but if one operand is **int** or **char** and the other is **float** or **double**, both operands are con-

verted to **double**. Thus if i and j are **int** and x is **float**,

```
(x+i)/j        converts i and j to float
x + i/j        does i/j integer, then converts
```

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts x to integer (truncating toward zero), and n to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is 'e' instead of 'E'. Thus:

```
pi = 3.14159;
large = 1.23456789e10;
```

printf will format floating point numbers: "%w.df" in the format string will print the corresponding variable in a field w digits wide, with d decimal places. An e instead of an f will produce exponential notation.

27. Horrors! goto's and labels

C has a **goto** statement and labels, so you can branch about the way you used to. But most of the time **goto**'s aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by **for/while**, **if/else**, and compound statements.

One use of **goto**'s with some legitimacy is in a program which contains a long loop, where a **while(1)** would be too extended. Then you might write

```
mainloop:
        ...
        goto mainloop;
```

Another use is to implement a **break** out of more than one level of **for** or **while**. **goto's** can only branch to labels within the same function.

28. Acknowledgements

I am indebted to a veritable host of readers who made valuable criticisms on several drafts of this tutorial. They ranged in experience from complete beginners through several implementors of C compilers to the C language designer himself. Needless to say, this is a wide enough spectrum of opinion that no one is satisfied (including me); comments and suggestions are still welcome, so that some future version might be improved.

References

C is an extension of B, which was designed by D. M. Ritchie and K. L. Thompson [4]. The C language design and UNIX implementation are the work of D. M. Ritchie. The GCOS version was begun by A. Snyder and B. A. Barres, and completed by S. C. Johnson and M. E. Lesk. The IBM version is primarily due to T. G. Peterson, with the assistance of M. E. Lesk.

[1] D. M. Ritchie, *C Reference Manual.* Bell Labs, Jan. 1974.

[2] M. E. Lesk & B. A. Barres, *The GCOS C Library.* Bell Labs, Jan. 1974.

[3] D. M. Ritchie & K. Thompson, *UNIX Programmer's Manual.* 5th Edition, Bell Labs, 1974.

[4] S. C. Johnson & B. W. Kernighan, *The Programming Language B.* Computer Science Technical Report 8, Bell Labs, 1972.

The C Programming Language — Reference Manual

Dennis M. Ritchie

Bell Laboratories, Murray Hill, New Jersey

This manual is reprinted, with minor changes, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
DEC VAX-11	8 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int        extern     else
char       register   for
float      typedef    do
double     static     while
struct     goto       switch
union      return     case
long       sizeof     default
short      break      entry
unsigned   continue
auto       if
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some

† UNIX is a Trademark of Bell Laboratories.

implementations also reserve the words `fortran` and `asm`.

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics which affect sizes are summarized in §2.6.

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. The digits 8 and 9 have octal value 10 and 11 respectively. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote ' and the backslash \, may be represented according to the following table of escape sequences:

newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
backslash	\	`\\`
single quote	'	`\'`
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in `"..."`. A string has type "array of characters" and storage class `static` (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte `\0` at the end of each string so that programs which scan the string can find its end. In a string, the double quote character " must be preceded by a \; in addition, the same escapes as described for character constants may be used. Finally, a \ and an immediately following newline are ignored.

2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

The VAX-11 is identical to the PDP-11 except that integers have 32 bits.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ $expression_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

4. What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double-precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will collectively be called *integral* types. **float** and **double** will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing a sequence of objects of various types;

unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from −128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value `-1`.

When a longer integer is converted to a shorter or to a `char`, it is truncated on the left; excess bits are simply discarded.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

> First, any operands of type `char` or `short` are converted to `int`, and any of type `float` are converted to `double`.

Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.
Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.
Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.
Otherwise, both operands must be `int`, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. All existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary-expression:
  identifier
  constant
  string
  (*expression*)
  primary-expression [*expression*]
  primary-expression (*expression-list*$_{opt}$)
  primary-lvalue . *identifier*
  primary-expression -> *identifier*

expression-list:
  expression
  expression-list , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int`; floating constants are `double`.

A string is a primary expression. Its type is originally "array of `char`"; but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a - and a >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
    `*` *expression*
    `&` *lvalue*
    `-` *expression*
    `!` *expression*
    `~` *expression*
    `++` *lvalue*
    `--` *lvalue*
    lvalue `++`
    lvalue `--`
    `(` *type-name* `)` *expression*
    `sizeof` *expression*
    `sizeof (` *type-name* `)`

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in an `int`. There is no unary + operator.

The result of the logical negation operator `!` is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The `~` operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x+=1`. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast.* Type names are described in §8.7.

The `sizeof` operator yields the size, in bytes, of i's operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be `float`.

7.4 Additive operators

The additive operators + and − group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression − *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the − operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same

array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

shift-expression:
  expression << expression
  expression >> expression

The value of E1<<E2 is E1 (interpreted as a bit pattern) left-shifted E2 bits; vacated bits are 0-filled. The value of E1>>E2 is E1 right-shifted E2 bit positions. The right shift is guaranteed to be logical (0-fill) if E1 is **unsigned**; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; **a<b<c** does not mean what it seems to.

relational-expression:
  expression < expression
  expression > expression
  expression <= expression
  expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
  expression == expression
  expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise AND operator

and-expression:
  expression & expression

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9 Bitwise exclusive OR operator

exclusive-or-expression:
  expression ^ expression

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10 Bitwise inclusive OR operator

inclusive-or-expression:
expression | *expression*

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11 Logical AND operator

logical-and-expression:
expression && *expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.12 Logical OR operator

logical-or-expression:
expression || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.13 Conditional operator

conditional-expression:
expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = *expression*
lvalue += *expression*
lvalue −= *expression*
lvalue *= *expression*
lvalue /= *expression*
lvalue %= *expression*
lvalue >>= *expression*
lvalue <<= *expression*
lvalue &= *expression*
lvalue ^= *expression*
lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left

preparatory to the assignment.

The behavior of an expression of the form `E1 op= E2` may be inferred by taking it as equivalent to `E1 = E1 op (E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
  expression **,** *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
  decl-specifiers declarator-list$_{opt}$ **;**

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
```
auto
static
extern
register
typedef
```

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int`, `char`, or pointer. One other restriction applies to register variables: the address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

8.2 Type specifiers

The type-specifiers are

type-specifier:
  `char`
  `short`
  `int`
  `long`
  `unsigned`
  `float`
  `double`
  struct-or-union-specifier
  typedef-name

The words `long`, `short`, and `unsigned` may be thought of as adjectives; the following combinations are acceptable.

```
short int
long int
unsigned int
long float
```

The meaning of the last is the same as `double`. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures and unions are discussed in §8.5; declarations with `typedef` names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
  init-declarator
  init-declarator , *declarator-list*

init-declarator:
  declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
  identifier
  (*declarator*)
  * *declarator*
  declarator ( )
  declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where T is a type-specifier (like int, etc.) and D1 is a declarator. Suppose this declaration makes the identifier have type "... T," where the "..." is empty if D1 is just a plain identifier (so that the type of x in "int x" is just int). Then if D1 has the form

```
*D
```

the type of the contained identifier is "... pointer to T."

If D1 has the form

```
D()
```

then the contained identifier has the type "... function returning T."

If D1 has the form

D [*constant-expression*]

or

```
D[]
```

then the contained identifier has type "... array of T." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures, unions or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer i, a pointer ip to an integer, a function f returning an integer, a function fip returning a pointer to an integer, and a pointer pfi to a function which returns an integer. It is especially useful to compare the last two. The binding of *fip() is *(fip()), so that the declaration suggests, and the same construction in an expression requires, the calling of a function fip, and then using indirection through the (pointer) result to yield an integer. In the declarator (*pfi)(), the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, x3d is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions x3d, x3d[i], x3d[i][j], x3d[i][j][k] may reasonably appear in an expression. The first three have type "array," the last has type int.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
  struct-or-union { *struct-decl-list* }
  struct-or-union identifier { *struct-decl-list* }
  struct-or-union identifier

struct-or-union:
  `struct`
  `union`

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , *struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

struct-declarator:
  declarator
  declarator : *constant-expression*
  : *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on the PDP-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

  `struct` *identifier* { *struct-decl-list* }
  `union` *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

  `struct` *identifier*
  `union` *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
      char tword[20];
      int count;
      struct tnode *left;
      struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the count field of the structure to which sp points;

```
s.left
```

refers to the left subtree pointer of the structure s; and

```
s.right->tword[0]
```

refers to the first character of the tword member of the right subtree of s.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and consists of an expression or a list of values nested in braces.

initializer:
    = *expression*
    = { *initializer-list* }
    = { *initializer-list* , }

initializer-list:
    expression
    initializer-list , *initializer-list*
    { *initializer-list* }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an ***aggregate*** (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes x as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for y begins with a left brace, but that for `y[0]` does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of y (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of `sizeof`) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
  type-specifier abstract-declarator

abstract-declarator:
  empty
  (*abstract-declarator*)
  * *abstract-declarator*
  abstract-declarator ( )
  abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

  (*abstract-declarator*)

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of 3 integers," "function returning pointer to integer," and "pointer to function returning an integer."

8.8 Typedef

Declarations whose "storage class" is **typedef** do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

> *typedef-name:*
>   *identifier*

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein become syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is "pointer to `int`," and that of `z` is the specified structure. `zp` is a pointer to such a structure.

`typedef` does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

> *expression ;*

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

> *compound-statement:*
>   { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }
>
> *declaration-list:*
>   *declaration*
>   *declaration declaration-list*
>
> *statement-list:*
>   *statement*
>   *statement statement-list*

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

> `if` (*expression*) *statement*
> `if` (*expression*) *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

9.4 While statement

The `while` statement has the form

> `while` (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The do statement has the form

> do *statement* `while` (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The `for` statement has the form

> `for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*

This statement is equivalent to

```
expression-1 ;
while (expression-2) {
        statement
        expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

> `switch` (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

> `case` *constant-expression* :

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`case` and `default` prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see `break`, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {          do {                   for (...) {
  ...                    ...                    ...
contin: ;              contin: ;              contin: ;
}                      } while (...);         }
```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function.

9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

identifier :

[illegible] the identifier as a label. The only use of a label is as a target of a goto. The [illegible] of a label is the current function, excluding any sub-blocks in which the same identifier has [illegible] redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as `while`.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ function-declarator function-body

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (parameter-list$_{opt}$)

parameter-list:
identifier
identifier , parameter-list

The function-body has the form

function-body:
declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
	int m;

	m = (a > b) ? a : b;
	return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; { ... } is the block giving the code for the statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

10.2 External data definitions

An external data definition has the form

> *data-definition:*
> *declaration*

The storage class of such data may be `extern` (which is the default) or `static`, but not `auto` or `register`.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
      auto int distance;
      ...
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance`†.

11.2 Scope of externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the `extern` keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

---

†It is agreed that the ice is thin here.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

`#define` *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

`#define` *identifier*`(` *identifier* `, ... ,` *identifier* `)` *token-string*

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

`#undef` *identifier*

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

`#include "`*filename*`"`

causes the replacement of that line by the entire contents of the file ***filename***. The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

`#include` <*filename*>

searches only the standard places, and not the directory of the source file.

`#include`'s may be nested.

12.3 Conditional compilation

A compiler control line of the form

`#if` *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

`#ifdef` *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. A control line of the form

`#ifndef` *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
#else
```

and then by a control line

```
#endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and an `#else` or, lacking an `#else`, the `#endif`, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

`#line` *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning ...", it is implicitly declared to be `extern`.

In an expression, an identifier followed by `(` and not already declared is contextually declared to be "function returning `int`".

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the `.` operator); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with `.` or `->`) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before `.`, and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a `->` is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of `g` might read

```
g(funcp)
int (*funcp)();
{
      ...
      (*funcp)();
      ...
}
```

Notice that f must be declared explicitly in the calling routine since its appearance in g(f) was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that E1[E2] is identical to *((E1)+(E2)). Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here x is a 3×5 array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression x[i], which is equivalent to *(x+i), x is first converted to a pointer as described; then i is converted to the type of x, which involves multiplying i by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an int or long is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a char pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

alloc must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to double; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is measured in bytes. chars have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word just to their right. Thus char pointers are measured in units of 2^{16} bytes; everything else is measured in units of 2^{18} machine words. double quantities and aggregates containing them must lie on an even word address (0 mod 2^{19}).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to short must be 0 mod 2, to int and float 0 mod 4, and to double 0 mod 8. Aggregates are aligned on the strictest boundary required by any of their constituents.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after case, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and sizeof expressions, possibly connected by the binary operators

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   <   >   <=   >=
```

or by the unary operators

```
-   ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Portability considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of register variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid register declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11, and VAX-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type int, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and VAX-11 and left-to-right on other machines. These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an int pointer to a char pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit-fields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

17. Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form =*op* instead of *op*= for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which actually decrements **x** since the = and the - are adjacent, but which might easily be intended to assign -1 to **x**.

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
int  x     = 1;
```

one used

```
int  x     1;
```

The change was made because the initialization

```
int  f     (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

18. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

18.1 Expressions

The basic expressions are:

expression:
    primary
    * *expression*
    & *expression*
    − *expression*
    ! *expression*
    ~ *expression*
    ++ *lvalue*
    −− *lvalue*
    lvalue ++
    lvalue −−
    `sizeof` *expression*
    (*type-name*) *expression*
    expression binop expression
    expression ? *expression* : *expression*
    lvalue asgnop expression
    expression , *expression*

primary:
    identifier
    constant
    string
    (*expression*)
    primary (*expression-list*$_{opt}$)
    primary [*expression*]
    lvalue . *identifier*
    primary −> *identifier*

lvalue:
    identifier
    primary [*expression*]
    lvalue . *identifier*
    primary −> *identifier*
    * *expression*
    (*lvalue*)

The primary-expression operators

    ()   []   .   −>

have highest priority and group left-to-right. The unary operators

    *   &   −   !   ~   ++   −−   `sizeof`   (*type-name*)

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```
*    /    %
+    -
>>   <<
<    >    <=    >=
==   !=
&
^
|
&&
||
?:
```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

The comma operator has the lowest priority, and groups left-to-right.

18.2 Declarations

declaration:
  decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

sc-specifier:
  `auto`
  `static`
  `extern`
  `register`
  `typedef`

type-specifier:
  `char`
  `short`
  `int`
  `long`
  `unsigned`
  `float`
  `double`
  struct-or-union-specifier
  typedef-name

init-declarator-list:
  init-declarator
  init-declarator , *init-declarator-list*

init-declarator:
  declarator initializer$_{opt}$

declarator:
  identifier
  (*declarator*)
  * *declarator*
  declarator ()
  declarator [*constant-expression*$_{opt}$]

struct-or-union-specifier:
    `struct` { *struct-decl-list* }
    `struct` *identifier* { *struct-decl-list* }
    `struct` *identifier*
    `union` { *struct-decl-list* }
    `union` *identifier* { *struct-decl-list* }
    `union` *identifier*

struct-decl-list:
    struct-declaration
    struct-declaration struct-decl-list

struct-declaration:
    type-specifier struct-declarator-list ;

struct-declarator-list:
    struct-declarator
    struct-declarator , *struct-declarator-list*

struct-declarator:
    declarator
    declarator : *constant-expression*
    : *constant-expression*

initializer:
    = *expression*
    = { *initializer-list* }
    = { *initializer-list* , }

initializer-list:
    expression
    initializer-list , *initializer-list*
    { *initializer-list* }

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (*abstract-declarator*)
    * *abstract-declarator*
    abstract-declarator ( )
    abstract-declarator [*constant-expression*$_{opt}$]

typedef-name:
    identifier

18.3 Statements

compound-statement:
    { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
    declaration
    declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*
`while` (*expression*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*
`switch` (*expression*) *statement*
`case` *constant-expression* : *statement*
`default` : *statement*
`break` ;
`continue` ;
`return` ;
`return` *expression* ;
`goto` *identifier* ;
identifier : *statement*
;

18.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
{ *declaration-list*$_{opt}$ *statement-list* }

data-definition:
`extern`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;
`static`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

18.5 Preprocessor

```
#define identifier token-string
#define identifier( identifier , ... , identifier ) token-string
#undef identifier
#include "filename"
#include <filename>
#if constant-expression
#ifdef identifier
#ifndef identifier
#else
#endif
#line constant identifier
```

Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978).

1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

enum-specifier

with syntax

enum-specifier:
  `enum` { *enum-list* }
  `enum` *identifier* { *enum-list* }
  `enum` *identifier*

enum-list:
  enumerator
  enum-list , *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, winedark };
...
enum color *cp, col;
```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were `int`.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

November 12, 1978

---

†UNIX is a Trademark of Bell Laboratories.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv)     /* echo arguments */
int argc;
char *argv[];
{
     int i;

     for (i = 1; i < argc; i++)
          printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if `prog` uses `getchar`,

then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using >: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
     int c;
     while ((c = getchar()) != EOF)
          if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
               putchar(c);
     exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the

program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character c on the file fp and returns c. getc and putc return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called stdin, stdout, and stderr. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. stdin, stdout and stderr are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type FILE * can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[];
{
	int c, i, inword;
	FILE *fp, *fopen();
	long linect, wordct, charct;
	long tlinect = 0, twordct = 0, tcharct = 0;

	i = 1;
	fp = stdin;
	do {
		if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
			fprintf(stderr, "wc: can't open %s\n", argv[i]);
			continue;
		}
		linect = wordct = charct = inword = 0;
		while ((c = getc(fp)) != EOF) {
			charct++;
			if (c == '\n')
				linect++;
			if (c == ' ' || c == '\t' || c == '\n')
				inword = 0;
			else if (inword == 0) {
				inword = 1;
				wordct++;
			}
		}
		printf("%7ld %7ld %7ld", linect, wordct, charct);
		printf(argc > 1 ? " %s\n" : "\n", argv[i]);
		fclose(fp);
		tlinect += linect;
		twordct += wordct;
		tcharct += charct;
	} while (++i < argc);
	if (argc > 2)
		printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
	exit(0);
}
```

The function fprintf is identical to printf, save that the first argument is a file pointer that specifies the file to be written.

The function fclose is the inverse of fopen; it breaks the connection between the file pointer and the external name that was established by fopen, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call fclose on an output file — it flushes the buffer in which putc is collecting output. (fclose is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

stderr is assigned to a program in the same way that stdin and stdout are. Output written on stderr appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on stderr instead of stdout so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function exit to terminate program execution. The argument of exit is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls fclose for each open output file, to flush out any buffered output, then calls a routine named _exit. The function _exit causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with putc, etc., is buffered (except to stderr); to force it out immediately, use fflush(fp).

fscanf is identical to scanf, except that its first argument is a file pointer (as with fprintf) that specifies the file from which the input comes; it returns EOF at end of file.

The functions sscanf and sprintf are identical to fscanf and fprintf, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for sscanf and into it for sprintf.

fgets(buf, size, fp) copies the next line from fp, up to and including a newline, into buf; at most size-1 characters are copied; it returns NULL at end of file. fputs(buf, fp) writes the string in buf onto file fp.

The function ungetc(c, fp) "pushes back" the character c onto the input stream fp; a subsequent call to getc, fscanf, etc., will encounter c. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and −1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define    BUFSIZE    512  /* best size for PDP-11 UNIX */

main()     /* copy input to output */
{
     char buf[BUFSIZE];
     int  n;

     while ((n = read(0, buf, BUFSIZE)) > 0)
          write(1, buf, n);
     exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define    CMASK      0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
     char c;

     return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define    CMASK      0377 /* for making char's > 0 */
#define    BUFSIZE    512

getchar() /* buffered version */
{
     static char     buf[BUFSIZE];
     static char     *bufp = buf;
     static int      n = 0;

     if (n == 0) {  /* buffer is empty */
          n = read(0, buf, BUFSIZE);
          bufp = buf;
     }
     return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns −1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and −1 if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
     int  f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
          error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
          error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
          error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
          if (write(f2, buf, n) != n)
               error("cp: write error", NULL);
     exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
     printf(s1, s2);
     printf("\n");
     exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
	lseek(fd, pos, 0);  /* get to pos */
	return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed

because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
	system("date");
	/* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember than `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where argp is an array of pointers to the arguments; the last pointer in the array must be NULL so execv can tell where the list ends. As with execl, filename is the file in which the program is found, and argp[0] is the name of the program. (This arrangement is identical to the argv array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use execl to invoke the shell sh, which then does all the work. Construct a string commandline that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /bin/sh. Its argument -c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl or execv. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc_id, the "process id." In one of these processes (the "child"), proc_id is zero. In the other (the "parent"), proc_id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
     execl("/bin/sh", "sh", "-c", cmd, NULL);     /* in child */
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
int status;

if (fork() == 0)
     execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to

return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int  fd[2];

stat = pipe(fd);
if (stat == -1)
     /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `fork`s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `execl`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define   READ 0
#define   WRITE     1
#define   tst(a, b) (mode == READ ? (b) : (a))
static    int  popen_pid;

popen(cmd, mode)
char *cmd;
int  mode;
{
     int p[2];

     if (pipe(p) < 0)
          return(NULL);
     if ((popen_pid = fork()) == 0) {
          close(tst(p[WRITE], p[READ]));
          close(tst(0, 1));
          dup(tst(p[READ], p[WRITE]));
          close(tst(p[READ], p[WRITE]));
          execl("/bin/sh", "sh", "-c", cmd, 0);
          _exit(1); /* disaster has occurred if we get here */
     }
     if (popen_pid == -1)
          return(NULL);
     close(tst(p[READ], p[WRITE]));
     return(tst(p[WRITE], p[READ]));
}
```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
     register r, (*hstat)(), (*istat)(), (*qstat)();
     int   status;
     extern int popen_pid;

     close(fd);
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     hstat = signal(SIGHUP, SIG_IGN);
     while ((r = wait(&status)) != popen_pid && r != -1);
     if (r == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     signal(SIGHUP, hstat);
     return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
     int onintr();

     if (signal(SIGINT, SIG_IGN) != SIG_IGN)
          signal(SIGINT, onintr);

     /* Process ... */

     exit(0);
}

onintr()
{
     unlink(tempfile);
     exit(1);
}
```

Why the test and the double call to signal? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the onintr routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that signal returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf   sjbuf;

main()
{
     int (*istat)(), onintr();

     istat = signal(SIGINT, SIG_IGN);    /* save original status */
     setjmp(sjbuf); /* save current stack position */
     if (istat != SIG_IGN)
          signal(SIGINT, onintr);

     /* main processing loop */
}
```

```
onintr()
{
     printf("\nInterrupt\n");
     longjmp(sjbuf);      /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
     if (intflag)
          /* EOF caused by interrupt */
     else
          /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
     execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr);  /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s) /* run command string s */
char *s;
{
     int status, pid, w;
     register int (*istat)(), (*qstat)();

     if ((pid = fork()) == 0) {
          execl("/bin/sh", "sh", "-c", s, 0);
          _exit(127);
     }
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     while ((w = wait(&status)) != pid && w != -1)
          ;
     if (w == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define   SIG_DFL   (int (*)())0
#define   SIG_IGN   (int (*)())1
```

References

[1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file

`stdout` The name of the standard output file

`stderr` The name of the standard error file

`EOF` is actually −1, and is the value returned by the read routines on end-of-file or error.

`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error

`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.

`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.

`getc, getchar, putc, putchar, feof, ferror, fileno`
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. Calls

```
FILE *fopen(filename, type) char *filename, *type;
```

opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL the attempt to open failed.

```
FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;
```

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, NULL is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`
returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer EOF is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`
acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`
`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but EOF is returned on error.

`fputc(c, ioptr) FILE *ioptr;`
acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`
The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`
Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`
returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`
is identical to `getc(stdin)`.

`putchar(c);`
is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`
reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or NULL if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`
writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`

The argument character c is pushed back on the input stream named by ioptr. Only one character may be pushed back.

printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;

printf writes on the standard output. fprintf writes on the named output stream. sprintf puts characters in the character array (string) named by s. The specifications are as described in section printf(3) of the *UNIX Programmer's Manual.*

scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;

scanf reads from the standard input. fscanf reads from the named input stream. sscanf reads from the character string supplied as s. scanf reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string format, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
reads nitems of data beginning at ptr from file ioptr. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
Like fread, but in the other direction.

rewind(ioptr) FILE *ioptr;
rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.

system(string) char *string;
The string is executed by the shell as if typed at the terminal.

getw(ioptr) FILE *ioptr;
returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer feof and ferror should be used. A "word" is 16 bits on the PDP-11.

putw(w, ioptr) FILE *ioptr;
writes the integer w on the named output stream.

setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char buf[BUFSIZ];
```

fileno(ioptr) FILE *ioptr;
returns the integer file descriptor associated with the file.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
The location of the next byte in the stream named by ioptr is adjusted. offset is a long integer. If ptrname is 0, the offset is measured from the beginning of the file; if ptrname is 1, the offset is measured from the current read or write pointer; if ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When

this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

`long ftell(ioptr) FILE *ioptr;`
The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`
The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`
allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`
allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available .

`cfree(ptr) char *ptr;`
Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine "core" files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

May 5, 1977

†UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX† with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (−) means ignore this argument as in:

adb − core

ADB has requests for examining locations in either file. The **?** request examines the contents of *objfile*, the / request examines the *corefile*. The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

†UNIX is a Trademark of Bell Laboratories.

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, −, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name;* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, '**D**' for long decimal, and '**F**' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	**Meaning**
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**
;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp*. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by **?** whereas the map for *core* file is referenced by **/**. Furthermore, a good rule of thumb is to use **?** for instructions and **/** for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp*. This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main.*

The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main*. Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the * instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.−10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f*, *g*, and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f*, *g*, and *h*. Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f*,*g*,*h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f*. Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h*, type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **$C** or the occurrence of a variable in the most recent call of a function. It is possible with the **$C** request, however, to print the stack frame starting at some address as **address$C.**

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out –

Breakpoints are set in the program as:

address:b [request]

The requests:

settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

$b

The display indicates a *count* field. A breakpoint is bypassed *count –1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

settab,5?ia

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the ? command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

:r

To delete a breakpoint, for instance the entry to the function *settab*, one types:

settab+4:d

To continue execution of the program from the breakpoint type:

:c

Once the program has stopped (in this case at the breakpoint for *fopen*), ADB requests can be used to display the contents of memory. For example:

$C

to display a stack trace, or:

tabs,3/8o

to print three lines of 8 locations each from the array called *tabs*. By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs*.

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

:c

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

ibuf+6/20c

When we continue the program with:

:c

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

tabpos+4:d

If the program is continued with:

:c

it resumes normal execution after ADB prints the message

a.out:running

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

:c

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

:c 0

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

settab+4:b settab,5?ia *

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

```
settab+4:b      settab,5?ia;0
getc+4,3:b      main.c?C;0
settab+4:b      settab,5?ia; ptab/o;0
```

Note that ;0 will set dot to zero and stop at the breakpoint.

stop after the third occurrence by typing:

getc+4,3:b main.c?C *

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

settab+4:b .,5?ia
fopen+4:b

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

settab+4:b settab,5?ia; ptab/o *

could be entered after typing the above requests.

Now the display of breakpoints:

$b

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f*, *g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

adb ex3 –

Suppose we enter the following breakpoints:

```
h+4:b   hcnt/d; h.hi/; h.hr/
g+4:b   gcnt/d; g.gi/; g.gr/
f+4:b   fcnt/d; f.fi/; f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b   fcnt/d; f.a/; f.b/; f.fi/
g+4:b   gcnt/d; g.p/; g.q/; g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

f+4:b   fcnt/d; f.a/"a="d; f.b/"b="d; f.fi/"fi="d

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  :r arg1 arg2 ... <infile >outfile

  This request kills any existing program under test and starts the *a.out* afresh.
- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.
- ADB allows a program to be entered at a specific address by typing:

  address:r
- The count field can be used to skip the first *n* breakpoints as:

  ,n:r

  The request:

  ,n:c

  may also be used for skipping the first *n* breakpoints when continuing a program.
- A program can be continued at an address different from the breakpoint by:

  address:c
- The program being debugged runs as a separate process and can be killed by:

  :k

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution

of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the ?* operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

$$b1 \leq A \leq e1 \Rightarrow \textbf{file address} = (A - b1) + f1$$
$$b2 \leq A \leq e2 \Rightarrow \textbf{file address} = (A - b2) + f2$$

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

<b

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

02000>b

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

<b,−1/4o4^8Cn

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b The base address of the data segment.

<b,−1 Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

4o Print 4 octal locations.

4^ Backup the current address 4 locations (to the original start of the field).

8C Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.

n Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, *dump*, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

=3n"C Stack Backtrace"

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero

thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072        58        #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a        0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write*, **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate*, **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

?l value

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

adb −w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

adb −w ex7 −
?l 'Th'
?W 'The '

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

?l 'Th'; ?s

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

adb a.out −
:s arg1 arg2
flag/w 1
:c

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if **?** is used for text (instructions) and **/** for data.

3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = 'T';
printf("debug 1 %s\n",charp);
        while(cc= *charp++)
                putc(cc,obuf);
        fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
          argc:      02
          argv:      0177762
          cc:        02124
$r
ps      0170010
pc      0204    ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:    0
_obuf:    0
_charp:   0124
_errno:   0
_fout:    0
$m
text map   `ex1'
b1 = 0             e1   = 02360          f1 = 020
b2 = 0             e2   = 02360          f2 = 020
data map   `core1'
b1 = 0             e1   = 03500          f1 = 02000
b2 = 0175400       e2   = 0200000        f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx        Nh@x&_
-
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756:        2
*main.argv/3o
0177762:        0177770 0177776 0177777
0177770/s
0177770:        a.out
*main.argv/3o
0177762:        0177770 0177776 0177777
*"/s
0177770:        a.out
 .=o
                0177770
 .-10/d
0177756:        2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
        x:          04452
        y:          04451
        hi:         ?
~g(04453,011124)
        p:          04453
        q:          011124
        gi:         04451
        gr:         ?
~f(02,04451)
        a:          02
        b:          04451
        fi:         011124
        fr:         04453
~h(04450,04447)
        x:          04450
        y:          04447
        hi:         04451
        hr:         02
~g(04451,011120)
        p:          04451
        q:          011120
        gi:         04447
        gr:         04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE     80
#define YES         1
#define NO          0
#define TABSP       8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n':/*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count   bkpt            command
1       ~tabpos+04
1       _getc+04
1       _fopen+04
1       ~settab+04
settab,5?ia
~settab:        jsr     r5,csv
~settab+04:     tst     -(sp)
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
settab,5?i
~settab:        jsr     r5,csv
                tst     -(sp)
                clr     0177770(r5)
                cmp     $0120,0177770(r5)
                blt     ~settab+076
:r
a.out: running
breakpoint      ~settab+04:     tst     -(sp)
settab+4:d
:c
a.out: running
breakpoint      _fopen+04:      mov     04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
        col:    01
        c:      0
        ptab:   03500
tabs,3/8o
03500:          01  0  0  0  0  0  0  0
                01  0  0  0  0  0  0  0
                01  0  0  0  0  0  0  0
```

Figure 6b: ADB output for C program of Figure 5

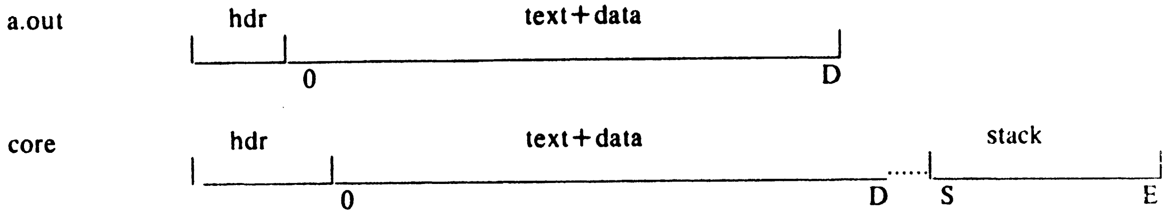
```
:c
a.out: running
breakpoint      _getc+04:     mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:          This    is      a test   of
:c
a.out: running
breakpoint      ~tabpos+04:   cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt            command
1       ~tabpos+04
3       _getc+04        main.c?C;0
1       _fopen+04
1       ~settab+04      settab,5?ia;ptab?o;0
~settab:        jsr     r5,csv
~settab+04:     bpt
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
0177766:        0177770
0177744:        @`
T0177744:       T
h0177744:       h
i0177744:       i
s0177744:       s
```

Figure 7: ADB output for C program with breakpoints

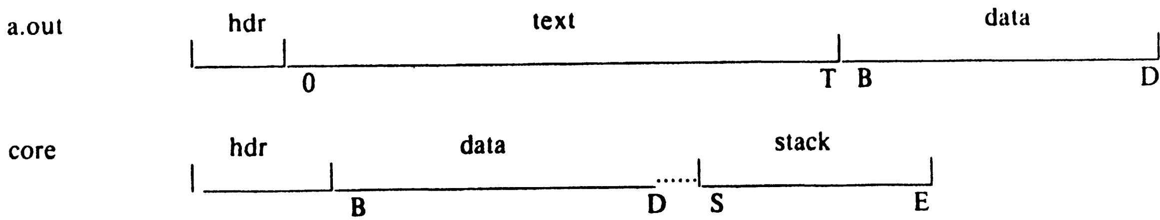
```
adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:       0
0177732:     214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:       0
0177746:     1
0177750:     1
0177732:     214
_gcnt:       0
0177726:     2
0177730:     3
0177712:     214
_hcnt:       0
0177706:     2
0177710:     1
0177672:     214
_fcnt:       1
0177666:     2
0177670:     3
0177652:     214
_gcnt:       1
0177646:     5
0177650:     8
0177632:     214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:       0
0177746:     a = 1
0177750:     b = 1
0177732:     fi = 214
_gcnt:       0
0177726:     p = 2
0177730:     q = 3
0177712:     gi = 214
_hcnt:       0
0177706:     x = 2
0177710:     y = 1
0177672:     hi = 214
_fcnt:       1
0177666:     a = 2
0177670:     b = 3
0177652:     fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

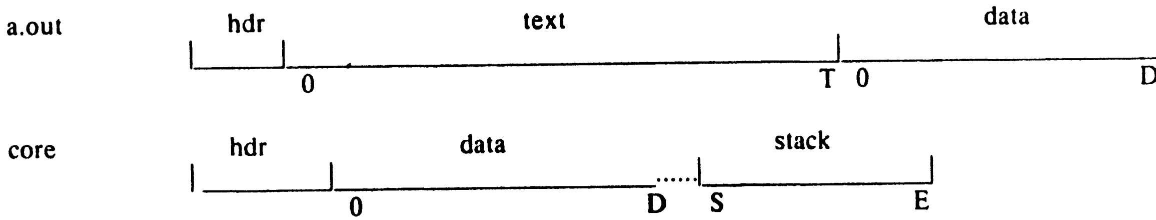
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D−B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407'
b1 = 0            e1     = 0256          f1 = 020
b2 = 0            e2     = 0256          f2 = 020
data map    `core407'
b1 = 0            e1     = 0300          f1 = 02000
b2 = 0175400      e2     = 0200000       f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410'
b1 = 0            e1     = 0200          f1 = 020
b2 = 020000       e2     = 020116   f2 = 0220
data map    `core410'
b1 = 020000       e1     = 020200   f1 = 02000
b2 = 0175400      e2     = 0200000       f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411'
b1 = 0            e1     = 0200          f1 = 020
b2 = 0            e2     = 0116          f2 = 0220
data map    `core411'
b1 = 0            e1     = 0200          f1 = 02000
b2 = 0175400      e2     = 0200000       f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    1234;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,-1/8ona
020000:      0       064124    071551    064440    020163    020141    064143    071141

_str1+016: 061541    062564    020162    072163    064562    063556    0    02

_number:
_number:   0710 0    02322040240    0    064124    071551    064440

_str2+06:  020163    064164    020145    062563    067543    062156    061440    060550

_str2+026: 060562    072143    071145    071440    071164    067151    0147 0

savr5+02:  0    0    0    0    0    0    0    0

<b,20/4o4^8Cn
020000:      0       064124    071551    064440    @`@`This i
           020163    020141    064143    071141    s a char
           061541    062564    020162    072163    acter st
           064562    063556    0    02    ring@`@`@b@`

_number:   0710 0    02322040240    H@a@`@`R@d @@
           0    064124    071551    064440    @`@`This i
           020163    064164    020145    062563    s the se
           067543    062156    061440    060550    cond cha
           060562    072143    071145    071440    racter s
           071164    067151    0147 0    tring@`@`@`
           0    0    0    0    @`@`@`@`@`@`@`@`
           0    0    0    0    @`@`@`@`@`@`@`@`
data address not found
<b,20/4o4^8t8cna
020000:      0       064124    071551    064440          This i
_str1+06:  020163    020141    064143    071141          s a char
_str1+016: 061541    062564    020162    072163          acter st
_str1+026: 064562    063556    0    02         ring
_number:
_number:   0710 0    02322040240         HR
_fpt+02:   0    064124    071551    064440          This i
_str2+06:  020163    064164    020145    062563          s the se
_str2+016: 067543    062156    061440    060550          cond cha
_str2+026: 060562    072143    071145    071440          racter s
_str2+036: 071164    067151    0147 0         tring
savr5+02:  0    0    0    0
savr5+012: 0    0    0    0
data address not found
<b,10/2b8t^2cn
020000:      0    0

_str1:     0124 0150    Th
           0151 0163    is
           040  0151     i
           0163 040     s
           0141 040     a
           0143 0150    ch
           0141 0162    ar
           0141 0143    ac
           0164 0145    te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?ut14cn

        Inode   Name
0:      652  .
        82   ..
        5971 cap.c
        5323 cap
        0    pp



adb /dev/src -
02000>b
?m<b
new map      '/dev/src'
b1 = 02000        e1      = 0100000000    f1 = 0
b2 = 0            e2      = 0             f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:      flags 073145
        links,uid,gid   0163 0164 0141
        size  0162 10356
        addr  28770    8236 25956    27766    25455    8236 25956    25206
        times1976 Feb 5 08:34:56  1975 Dec 28 10:55:15

02040:      flags 024555
        links,uid,gid   012  0163 0164
        size  0162 25461
        addr  8308 30050    8294 25130    15216    26890    29806    10784
        times1976 Aug 17 12:16:51 1976 Aug 17 12:16:51

02100:      flags 05173
        links,uid,gid   011  0162 0145
        size  0147 29545
        addr  25972    8306 28265    8308 25642    15216    2314 25970
        times1977 Apr 2 08:58:01  1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a) formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

> *name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
*n***t**	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
−	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*	contents of location
−	integer negate

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt
Bell Laboratories
Murray Hill, New Jersey 07974

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

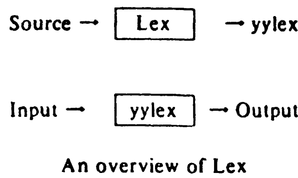
Table of Contents

1. Introduction. 1
2. Lex Source. 3
3. Lex Regular Expressions. 3
4. Lex Actions. 5
5. Ambiguous Source Rules. 7
6. Lex Source Definitions. 8
7. Usage. 8
8. Lex and Yacc. 9
9. Examples. 10
10. Left Context Sensitivity. 11
11. Character Set. 12
12. Summary of Source Format. 12
13. Caveats and Bugs. 13
14. Acknowledgments. 13
15. References. 13

1 Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to



An overview of Lex

Figure 1

write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$   ;
```

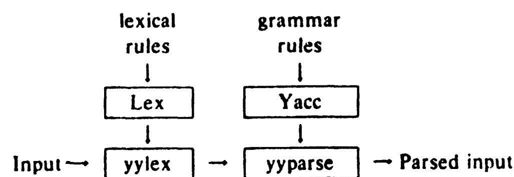
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$   ;
[ \t]+    printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time



Lex with Yacc

Figure 2

taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the ***actions*** to be performed as each regular expression is found) are gathered as cases of a switch (in C) or branches of a computed GOTO (in Ratfor). The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ***ab*** and another for ***abcdefg***, and the input stream is ***abcdefh***, Lex will recognize ***ab*** and leave the input pointer just before ***cd***. . . Such backup is more costly than the processing of simpler languages.

2 Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second **%%** is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the ***rules*** represent the user's control decisions; they are a table, in which the left column contains ***regular expressions*** (see section 3) and the right column contains ***actions***, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string ***integer*** in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function ***printf*** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour       printf("color");
mechanise    printf("mechanize");
petrol       printf("gas");
```

would be a start. These rules are not quite enough, since the word ***petroleum*** would become ***gaseum***; a way of dealing with this will be described later.

3 Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string ***integer*** wherever it appears and the expression

```
a57D
```

looks for the string ***a57D***.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string ***xyz++*** when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above

expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction *[abj* matches a single character, which may be ***a***, ***b***, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – character indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40-\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator *?* indicates an optional element of an expression. Thus

ab?c

matches either ***ac*** or ***abc***.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

*a**

is any number of consecutive *a* characters, including zero; while

a+

is one or more instances of ***a***. For example,

[a-z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]*

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab|cd)

matches either ***ab*** or ***cd***. Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab|cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab|cd+)?(ef)*

matches such strings as ***abefef***, ***efefef***, ***cdef***, or ***cddd***; but not ***abc***, ***abcd***, or ***abcdef***.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and ***$***. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is ***$***, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string ***ab***, but only if followed by ***cd***. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by ***start conditions*** as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named ***digit*** and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of ***a***.

Finally, initial **%** is special, being the separator for Lex source segments.

4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, *;* as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named ***yytext***. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

will print the string in ***yytext***. The C function ***printf*** accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in ***yytext***. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches ***read*** it will normally match the instances of ***read*** contained in ***bread*** or ***readjust***, to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count ***yyleng*** of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+    {words++; chars += yyleng;}
```

which accumulates in ***chars*** the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

in C or

```
yytext(yyleng)
```

in Ratfor.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, ***yymore()*** can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in ***yytext***. Second, ***yyless (n)*** may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument ***n*** indicates the number of characters in ***yytext*** to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng-1] == '\\')
               yymore();
           else
               ... normal user processing
           }
```

which will, when faced with a string such as ***"abc\"def"*** first match the five characters ***"abc***; then the call to ***yymore()*** will cause the next part of the string, ***"def***, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function ***yyless()*** might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=-[a-zA-Z]   {
             printf("Operator (=-) ambiguous\n");
             yyless(yyleng-1);
             ... action for =- ...
             }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]   {
             printf("Operator (=-) ambiguous\n");
             yyless(yyleng-2);
             ... action for = ...
             }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) ***input()*** which returns the next input character;
2) ***output(c)*** which writes the character ***c*** on the output; and
3) ***unput(c)*** pushes the character ***c*** back onto the input stream to be read later by ***input()***.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. There is another important routine in Ratfor, named ***lexshf***, which is described below under "Character Set". These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by ***input*** must mean end of file; and the relationship between ***unput*** and ***input*** must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in **+ * ?** or **$** or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is ***yywrap()*** which is called whenever Lex reaches an end-of-file. If ***yywrap*** returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a ***yywrap*** which arranges for new input and returns 0. This instructs Lex to continue processing. The default ***yywrap*** always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through ***yywrap***. In fact, unless a private version of ***input()*** is supplied a file containing nulls cannot be handled, since a value of 0 returned by ***input*** is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, ***input***,

output, unput, yywrap, and ***lexshf,*** are defined as integer functions. This requires ***input*** and ***yywrap*** to be called with arguments. One dummy argument is supplied and ignored.

5 Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

to be given in that order. If the input is ***integers***, it is taken as an identifier, because ***[a-z]+*** matches 8 characters while ***integer*** matches only 7. If the input is ***integer***, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. ***int***) will not match the expression ***integer*** and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after ***'first'***. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like ***[.\n]+*** or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she    s++;
he     h++;
\n     |
.      ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally ***not*** recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.      ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+    { ... ; REJECT;}
a[cd]+    { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string ***accb*** matches the first rule for four characters and then the second rule for three characters. In contrast, the input ***accd*** agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
\n            ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                   [0-9]
E                   [DEde][-+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}*({E})?   |
{D}*"."{D}+({E})?   |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named ***lex.yy.c*** for a C host language source and ***lex.yy.r*** for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file ***%R***.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file ***%C***.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of ***input*** removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll.* So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text.* The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a. Edit the Ratfor program.
   1. Remove all tabs.
   2. Change all lower case letters to upper case letters.
   3. Convert the file to an 80-column card image file.

b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.

c. Compile the Fortran.

d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. ***Warning:*** Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll -lS
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9 Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
            int k;
[0-9]+      {
            scanf(-1, yytext, "%d", &k);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

to do just that. The rule [0-9]+ recognizes strings of digits; *scanf* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
-?[0-9]+                {
                        scanf(-1, yytext, "%d", &k);
                        printf("%d", k%7 == 0 ? k+3 : k);
                        }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a    [aA]
b    [bB]
c    [cC]
...
z    [zZ]
```

An additional class recognizes white space:

```
W    [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' | *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'-'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}       |
{d}{l}{o}{g}10     |
{d}{m}{i}{n}1      |
{d}{m}{a}{x}1      {
                   yytext[0] =+ 'a' - 'd';
                   ECHO;
                   }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10 Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as *$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a              {ECHO; BEGIN AA;}
^b              {ECHO; BEGIN BB;}
^c              {ECHO; BEGIN CC;}
\n              {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11 Character Set.

The programs generated by Lex handle character I/O only through the routines ***input, output,*** and ***unput.*** Thus the character representation provided in these routines is accepted by Lex and employed to return values in ***yytext.*** For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine ***lexshf*** is called to change the representation delivered by ***input*** into a right-adjusted integer. If the user changes the I/O library, the routine ***lexshf*** should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter ***a*** as in the Fortran value ***1Ha*** while in C the letter ***a*** is represented as the character constant ***'a'.*** If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

```
%T
 1    Aa
 2    Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

Sample character table.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace ***input, output, unput*** and ***lexshf*** but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave ***input*** and ***output*** as routines that read with 80A1 format, but replace ***lexshf*** by a table lookup routine.

12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

%S name1 name2 ...

5) Character set tables, in the form

%T
number space character-string
...
%T

6) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) Changes to internal array sizes, in the form

%*x* *nnn*

where ***nnn*** is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

<table>
<tr><td>x</td><td>the character "x"</td></tr>
<tr><td>"x"</td><td>an "x", even if x is an operator.</td></tr>
<tr><td>\x</td><td>an "x", even if x is an operator.</td></tr>
<tr><td>[xy]</td><td>the character x or y.</td></tr>
<tr><td>[x-z]</td><td>the characters x, y or z.</td></tr>
<tr><td>[^x]</td><td>any character but x.</td></tr>
<tr><td>.</td><td>any character but newline.</td></tr>
<tr><td>^x</td><td>an x at the beginning of a line.</td></tr>
<tr><td><y>x</td><td>an x when Lex is in start condition y.</td></tr>
<tr><td>x$</td><td>an x at the end of a line.</td></tr>
<tr><td>x?</td><td>an optional x.</td></tr>
<tr><td>x*</td><td>0,1,2, ... instances of x.</td></tr>
<tr><td>x+</td><td>1,2,3, ... instances of x.</td></tr>
<tr><td>x|y</td><td>an x or a y.</td></tr>
<tr><td>(x)</td><td>an x.</td></tr>
<tr><td>x/y</td><td>an x but only if followed by y.</td></tr>
<tr><td>{xx}</td><td>the translation of xx from the definitions section.</td></tr>
<tr><td>x{m,n}</td><td>m through n occurrences of x</td></tr>
</table>

13 Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15 References.

1. B. W. Kernighan and D. M. Ritchie, ***The C Programming Language,*** Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, ***Ratfor: A Preprocessor for a Rational Fortran,*** Software — Practice and Experience, 5, pp. 395-496 (1975).
3. S. C. Johnson, ***Yacc: Yet Another Compiler Compiler,*** Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, ***Efficient String Matching: An Aid to Bibliographic Search,*** Comm. ACM **18**, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, ***QED Text Editor,*** Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c*

and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   −lS   −o  prog

x.o  y.o :   defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc −c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   −lS   −o  prog
x.o :  x.c  defs
        cc   −c  x.c
y.o :  y.c  defs
        cc   −c  y.c
z.o :  z.c
        cc   −c  z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup"

might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. `$$` is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: `$*`, `$@`, `$?`, and `$<`. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
  . . .
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex ("−ll") and the Standard ("−lS") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

---

†UNIX is a Trademark of Bell Laboratories.

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[*(tab)* commands] [# . . .]
. . .

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "−i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "−" denotes the standard input. If there are no "−f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

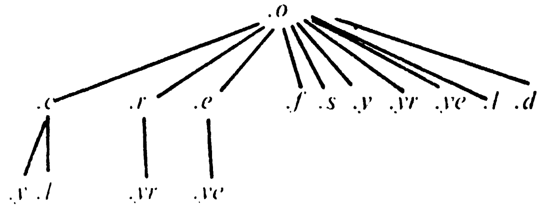
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

make CC=newcc

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

make "CFLAGS= −O"

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command
P = und −3 | opr −r2        # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= −lS
LINT = lint −p
CFLAGS = −O

make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) −o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES)     # print recently changed files
        pr $? | $P
        touch print

test:
        make −dp | grep −v TIME >1zap
        /usr/bin/make −dp | grep −v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint :  dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  −c version.c
cc  −c main.c
cc  −c doname.c
cc  −c misc.c
cc  −c files.c
cc  −c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  −c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o gram.o  −lS  −o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits

results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
        or
make print "P=  cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "−n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "−t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("−d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, "Lex — A Lexical Analyzer Generator", Computing Science Technical Report #39, October 1975.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "−r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc −r
YACCE=yacc −e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as −
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
        $(CC) $(CFLAGS) −c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) −c $<
.s.o :
        $(AS) −o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) −c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

July 26, 1978

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction and Usage

Suppose there are two C[1] source files, *file1*.c and *file2.c*, which are ordinarily compiled and loaded together. Then the command

    lint file1.c file2.c

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

    lint −p file1.c file2.c

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **−p** by **−h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **−hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **−x** flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **−v** option is available to suppress the printing of complaints about unused arguments. When **−v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **−u** flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while(1)** and **for(;;)** as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The **−O** flag [illegible]

compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the **−b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in "working" programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the −> be a pointer to structure, the left operand of the . be a structure, and the right operand of

these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char, short, int,** and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **−c** flag controls the printing of comments about casts. When **−c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from −128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
	...
if( (c = getchar()) < 0 ) ....
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **−a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **−h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the * does nothing; this provokes the message "null effect" from *lint.* The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the **−h** flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =−, . . .) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =-  1 ;
```

or

```
a =  -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, −=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned

operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

```
int x ( −1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = −1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **−p** or **−h** flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

will draw the complaint:

```
warning: i evaluation order undefined
```

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[4,5] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compiler. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file

which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint.*

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a.* Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a.* When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the **−p** flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* **−p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

†UNIX is a Trademark of Bell Laboratories.

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The −v flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

```
−ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are

pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.* **57**(6) pp. 2021-2048 (1978).
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* (January 1978).

Appendix: Current Lint Options

The command currently has the form

lint [−options] files... library-descriptors...

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable **break** statements.

x Report unused external declarations

a Report assignments of **long** to **int** or shorter.

c Complain about questionable casts

n No library checking is done

s Same as **h** (for historical reasons)

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

        . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a

month_name was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar)*

rules, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'    newline
'\r'    return
'\''    single quote "'"
'\\'    backslash "\"
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  "xxx" in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A   :   B C D ;
A   :   E F ;
A   :   G ;
```

can be given to Yacc as

```
A   :   B C D
    |   E F
    |   G
    ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker.* If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A     :     '(' B ')'
                  {     hello( 1, "abc" ); }
```

and

```
XXX   :     YYY ZZZ
                  {     printf("a message\n");
                        flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
        { $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :       '(' expr ')'            { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A       :       B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                        { $$ = 1; }
                C
                        { x = $2; y = $3; }
        ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                        { $$ = 1; }
        ;

A       :       B $ACT C
                        { x = $2; y = $3; }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :       expr '+' expr
                        { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval.*

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error

handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
        .       reduce 18
```

refers to *grammar rule* 18, while the action

```
        IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
    A       :       x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
        A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme :     sound place
      ;
sound :     DING DONG
      ;
place :     DELL
      ;
```

When Yacc is invoked with the −v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_ (1)

        . reduce 1

state 5
        place : DELL_ (3)

        . reduce 3

state 6
        sound : DING DONG_ (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

DING DONG DELL

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is

"shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound  :  DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound   goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr    :       expr '−' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  −  expr  −  expr
```

the rule allows this input to be structured as either

```
( expr  −  expr )  −  expr
```

or as

```
expr  −  ( expr  −  expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr  −  expr  −  expr
```

When the parser has read the second expr, the input that it has seen:

```
expr  −  expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

```
    − expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
    expr − expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
    expr − expr − expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
    expr − expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
    expr − expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict.* It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict.* Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule.*

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
    stat    :    IF '(' cond ')' stat
            |    IF '(' cond ')' stat ELSE stat
            ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
      IF ( C2 ) S1
      }
ELSE S2
```

or

```
IF ( C1 ) {
      IF ( C2 ) S1
      ELSE S2
      }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (−v) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
stat : IF ( cond ) stat_        (18)
stat : IF ( cond ) stat_ELSE stat

ELSE    shift 45
        reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly.in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2,3,4] might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and

construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '−'
%left '*' '/'

%%

expr    :       expr '+' expr
        |       expr '−' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '−' expr     %prec '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :    error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :    error '\n'  {  printf( "Reenter last line: " );  }  input
                         {       $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :    error '\n'
                  {      yyerrok;
                         printf( "Reenter last line: " );   }
             input
                  {      $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some

sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;  }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **−ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
        name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
        list    :       item
                |       list  ','  item
                ;
```

and

```
        seq     :       item
                |       seq  item
                ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
        seq     :       item
                |       item  seq
                ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ... other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj noun verb adj noun
                        {  look at the sentence . . .  }
        ;

adj     :       THE             {       $$ = THE;  }
        |       YOUNG           {       $$ = YOUNG;  }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG;  }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                        }
                                $$ = CRONE;
                                }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 − see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa { $<intval>$ = 3; } bbb
                        {       fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* (January 1978).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%    /* beginning of rules section */

list    :    /* empty */
        |    list stat '\n'
        |    list error '\n'
                  {    yyerrok; }
        ;

stat    :    expr
                  {    printf( "%d\n", $1 ); }
        |    LETTER '=' expr
                  {    regs[$1] = $3; }
        ;

expr    :    '(' expr ')'
                  {    $$ = $2; }
        |    expr '+' expr
                  {    $$ = $1 + $3; }
        |    expr '-' expr
                  {    $$ = $1 - $3; }
```

```
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number: DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
        /* returns LETTER for a lower case letter, yylval = 0 through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal) followed by colon */
%token  NUMBER                  /* [0-9]+ */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK {  In this action, eat up the rest of the file  }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
```

```
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty' */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where *x* is less than or equal to *y*. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef.* The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start   lines

%union   {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG        /* indices into dreg, vreg arrays */

%token <dval> CONST            /* floating point constant */

%type <dval> dexp        /* expression */

%type <vval> vexp        /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS     /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                {       vreg[$1] = $3; }
        |       error '\n'
                {       yyerrok; }
        ;

dexp    :       CONST
        |       DREG
                {       $$ = dreg[$1]; }
        |       dexp '+' dexp
                {       $$ = $1 + $3; }
        |       dexp '-' dexp
                {       $$ = $1 - $3; }
        |       dexp '*' dexp
                {       $$ = $1 * $3; }
        |       dexp '/' dexp
                {       $$ = $1 / $3; }
        |       '-' dexp        %prec UMINUS
                {       $$ = - $2; }
        |       '(' dexp ')'
                {       $$ = $2; }
        ;

vexp    :       dexp
                {       $$.hi = $$.lo = $1; }
        |       '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |       VREG
                {       $$ = vreg[$1];  }
        |       vexp '+' vexp
                {       $$.hi = $1.hi + $3.hi;
                        $$.lo = $1.lo + $3.lo;  }
        |       dexp '+' vexp
                {       $$.hi = $1 + $3.hi;
                        $$.lo = $1 + $3.lo;  }
        |       vexp '-' vexp
                {       $$.hi = $1.hi - $3.lo;
                        $$.lo = $1.lo - $3.hi;  }
        |       dexp '-' vexp
                {       $$.hi = $1 - $3.lo;
                        $$.lo = $1 - $3.hi;  }
        |       vexp '*' vexp
                {       $$ = vmul( $1.lo, $1.hi, $3 ); }
        |       dexp '*' vexp
                {       $$ = vmul( $1, $1, $3 ); }
        |       vexp '/' vexp
                {       if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = -$2.lo;  $$.lo = -$2.hi;  }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;

%%

# define BSZ 50         /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );  /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );  /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
		else ungetc( c, stdin );   /* push back last char read */
		yylval.dval = atof( buf );
		return( CONST );
		}
	return( c );
	}

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
	/* returns the smallest interval containing a, b, c, and d */
	/* used by *, / routines */
	INTERVAL v;

	if( a>b ) { v.hi = a;  v.lo = b; }
	else { v.hi = b;  v.lo = a; }

	if( c>d ) {
		if( c>v.hi ) v.hi = c;
		if( d<v.lo ) v.lo = d;
		}
	else {
		if( d>v.hi ) v.hi = d;
		if( c<v.lo ) v.lo = c;
		}
	return( v );
	}

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
	return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
	}

dcheck( v ) INTERVAL v; {
	if( v.hi >= 0. && v.lo <= 0. ){
		printf( "divisor interval contains 0.\n" );
		return( 1 );
		}
	return( 0 );
	}

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
	return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
	}
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes “"”.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash “\” may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.