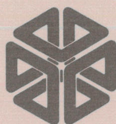
IRIS-4D Programmer's Guide

Volume I

IRIS-4D Series



SiliconGraphics
Computer Systems

Document number: 007-0601-010

IRIS-4D Programmer's Guide

Volume I

Version 1.1

Document Number 007-0601-010
(Includes *IRIS-4D Programmer's Guide Update,*
Document Number 007-0601-012)

5/90

Technical Publications:

Marcia Allen
Wendy Ferguson
Melissa Heinrich
Anne Wilson

Engineering:

Greg Boyd
Dave Ciemiewicz

© Copyright 1990, Silicon Graphics, Inc. - All rights reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc., and is protected by Federal copyright law. The contents of this document may not be disclosed to third parties, copied or duplicated in any form, in whole or in part, without the express written permission of Silicon Graphics, Inc.

U.S. Government Limited Rights

Use, duplication or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (b) (2) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Contractor/manufacturer is Silicon Graphics Inc., 2011 Shoreline Blvd., Mountain View, CA 94039-7311.

IRIS-4D Programmer's Guide
Version 1.1
Document Number 007-0601-010
(Includes *IRIS-4D Programmers's Guide Update,* Document Number 007-0601-012)

Silicon Graphics, Inc.
Mountain View, California

The words IRIX, IRIS, Geometry Link, Geometry Partners, Geometry Engine and Geometry Accelerator are trademarks of Silicon Graphics, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Contents

Introduction

- Purpose .... i-xiii
  - Audience and Prerequisite Knowledge .... i-xiii
  - Organization .... i-xiii
  - The C Connection .... i-xiv
  - Hardware/Software Dependencies .... i-xiv
  - Notation Conventions .... i-xiv
  - Command References .... i-xv
  - Information in the Examples .... i-xvi

1. Programming in a UNIX System Environment

- The UNIX System Environment 1-1
  - The UNIX Idea 1-1
  - The UNIX Philosophy 1-3
- UNIX System Tools 1-4
  - Tools Covered and Not Covered in This Guide 1-4
  - The Shell as a Prototyping Tool 1-5
- Three Programming Environments 1-6
  - Single-User Programming 1-6
  - Application Programming 1-7
  - Systems Programming 1-7
- Summary 1-8

2. Programming Basics

- Programming in a UNIX Environment 2-1
- Choosing a Programming Language 2-2
  - Supported Languages in the UNIX Environment 2-2
    - C Language 2-3
    - FORTRAN 2-4
    - Assembly Language 2-4
  - Special-Purpose Languages 2-4
    - awk 2-4
    - lex 2-5
    - yacc 2-5
    - M4 2-6
    - bc and dc 2-6

curses 2-6
Compiling and Link Editing 2-7
Compiling C Programs 2-7
Compiling FORTAN Programs 2-7
Compiler Diagnostic Messages 2-7
Link Editing 2-8
The UNIX System/Language Interface 2-9
Using C to Illustrate the Interface 2-9
Passing Arguments to a Program 2-9
System Calls and Subroutines 2-13
Categories of System Calls and Subroutines 2-13
Where the Manual Pages Can Be Found 2-24
Using System Calls and Subroutines in C Programs 2-24
Header Files and Libraries 2-30
Object File Libraries 2-31
Input and Output 2-32
Three Files to Handle Standard I/O Streams 2-32
Named Files 2-33
The UNIX System and Low-level I/O 2-34
System Calls for Environment or Status Information 2-34
Processes 2-36
system(3S) 2-38
exec(2) 2-38
fork(2) 2-39
Pipes 2-41
Error Handling 2-42
Signals and Interrupts 2-43
Analysis and Debugging 2-45
Sample Program 2-45
cflow 2-50
ctrace 2-53
cxref 2-57
lint 2-63
prof 2-64
size 2-65
strip 2-66
Program-Organizing Utilities 2-67
The make Command 2-67

The Archive File 2-68
Use of SCCS by Single-User Programmers 2-74

3. Application Programming

Application Programming Objectives 3-1
Application Environment Characteristics 3-2
Numbers 3-2
Portability 3-2
Documentation 3-2
Language Selection 3-4
Influences 3-4
Special-Purpose Languages 3-5
The **awk** Utility 3-5
Using **awk** 3-6
The **lex** and **yacc** Utilities 3-6
Using **lex** 3-7
Using **yacc** 3-8
Advanced Programming Tools 3-10
Memory Management 3-10
File and Record Locking 3-11
How File and Record Locking Works 3-12
lockf 3-14
Interprocess Communications 3-14
IPC **get** Calls 3-15
IPC **ctl** Calls 3-15
IPC **op** Calls 3-16
Programming Terminal Screens 3-16
curses 3-17
Programming Support Tools 3-18
Link Edit Command Language 3-18
Common Object File Format 3-18
Libraries 3-19
The Object File Library 3-19
Common Object File Interface
Macros (**ldfcn.h**) 3-22
The Math Library 3-23
A Basic Lesson on Debugging 3-26
How Does **edge** Work? 3-26
About the edge Environment 3-27
lint as a Portability Tool 3-27

Project Control Tools 3-29
make 3-29
SCCS 3-30
liber, A Library System 3-32

4. nawk

Introduction 4-1
Basic **nawk** 4-2
Program Structure 4-2
Usage 4-3
Fields 4-3
Printing 4-4
Formatted Printing....4-5
Simple Patterns 4-6
Simple Actions....4-7
Built-in Variables 4-7
User-defined Variables 4-8
Functions 4-8
A Handful of Useful One-liners....4-8
Error Messages 4-10
Patterns....4-11
BEGIN and **END** 4-11
Relational Expressions....4-12
Regular Expressions 4-13
Combinations of Patterns....4-16
Pattern Ranges 4-17
Actions 4-18
Built-in Variables 4-18
Arithmetic 4-19
Strings and String Functions 4-21
Field Variables....4-25
Number or String?....4-25
Control Flow Statements....4-27
Arrays 4-29
User-Defined Functions 4-31
Some Lexical Conventions....4-32
Output 4-33
The **print** Statement....4-33
Output Separators 4-33
The **printf** Statement....4-34

Output into Files 4-35
Output into Pipes 4-36
Input 4-38
Files and Pipes 4-38
Input Separators 4-38
Multi-line Records 4-39
The **getline** Function 4-39
Command-line Arguments 4-41
Using **nawk** with Other Commands and the Shell 4-43
The **system** Function 4-43
Cooperation with the Shell 4-43
Example Applications 4-46
Generating Reports 4-46
Additional Examples 4-48
Word Frequencies 4-48
Accumulation 4-48
Random Choice 4-49
Shell Facility 4-49
Form-letter Generation 4-49
nawk Summary 4-51
Command Line 4-51
Patterns 4-51
Control Flow Statements 4-51
Input-output 4-52
Functions 4-52
String Functions 4-53
Arithmetic Functions 4-53
Operators (Increasing Precedence) 4-54
Regular Expressions (Increasing Precedence) 4-54
Built-in Variables 4-55
Limits 4-55
Initialization, Comparison, and Type Coercion 4-56

5. lex

An Overview of **lex** Programming 5-1
Writing **lex** Programs 5-3
The Fundamental **lex** Rules 5-3
Specifications 5-3
Actions 5-5
Advanced **lex** Usage 5-6

Some Special Features 5-7
Definitions 5-11
Subroutines 5-12
Using **lex** with **yacc** 5-13
Running **lex** Under the UNIX System 5-16

6.yacc
An Overview of **yacc** Programming 6-1
Basic Specifications 6-4
Actions 6-6
Lexical Analysis 6-9
Parser Operation 6-12
Ambiguity and Conflicts 6-17
Precedence 6-22
Error Handling 6-26
The **yacc** Environment 6-30
Hints for Preparing Specifications 6-32
Input Style 6-32
Left Recursion 6-32
Lexical Tie-Ins 6-33
Reserved Words 6-35
Advanced **yacc** Features 6-36
Simulating **error** and **accept** in Actions 6-36
Accessing Values in Enclosing Rules 6-36
Support for Arbitrary Value Types 6-38
yacc Input Syntax 6-39
Examples 6-43
1. A Simple Example 6-43
2. An Advanced Example 6-47

7. File and Record Locking
An Overview of File and Record Locking 7-1
Terminology 7-2
File Protection 7-4
Opening a File for Record Locking 7-4
Setting a File Lock 7-5
Setting and Removing Record Locks 7-8
Getting Lock Information 7-13
Deadlock Handling 7-16
Selecting Advisory or Mandatory Locking 7-17

Mandatory Locking 7-18
Record Locking and Future UNIX Releases 7-18

8. Interprocess Communication
An Overview of Inter-Process Communication 8-1
Messages 8-2
Getting Message Queues 8-6
Using **msgget** 8-7
Example Program 8-12
Controlling Message Queues 8-16
Using **msgctl** 8-16
Example Program 8-18
Operations for Messages 8-24
Using **msgop** 8-24
Example Program 8-26
Semaphores 8-37
Using Semaphores 8-39
Getting Semaphores 8-42
Using **semget** 8-42
Example Program 8-46
Controlling Semaphores 8-50
Using **semctl** 8-51
Example Program 8-52
Operations on Semaphores 8-64
Using **semop** 8-64
Example Program 8-66
Shared Memory 8-72
Using Shared Memory 8-73
Getting Shared Memory Segments 8-77
Using **shmget** 8-77
Example Program 8-81
Controlling Shared Memory 8-85
Using **shmctl** 8-85
Example Program 8-87
Operations for Shared Memory 8-96
Using **shmop** 8-96
Example Program 8-97

9. curses/terminfo
The Terminal Information Utilities Package 9-1

What is **curses?** 9-2
What is **terminfo?** 9-3
How **curses** and **terminfo** Work Together 9-5
Other Components of the Terminal Information Utilities 9-5
Working with **curses** Routines 9-7
What Every **curses** Program Needs 9-7
The Header File **<curses.h>** 9-7
The Routines **initscr()**, **refresh()**, **endwin()** 9-8
Compiling a **curses** Program 9-10
Running a **curses** Program 9-10
More about **initscr()** and Lines and Columns 9-11
More about **refresh()** and Windows 9-11
Getting Simple Output and Input 9-13
Output 9-13
Input 9-26
Controlling Output and Input 9-34
Output Attributes 9-34
Bells, Whistles, and Flashing Lights 9-38
Input Options 9-39
Building Windows and Pads 9-44
Output and Input 9-44
The Routines **wnoutrefresh()** and **doupdate()** 9-45
New Windows 9-46
Using Advanced **curses** Features 9-50
Routines for Drawing Lines and Other Graphics 9-50
Routines for Using Soft Labels 9-52
Working with More than One Terminal 9-53
Working with **terminfo** Routines 9-55
What Every **terminfo** Program Needs 9-55
Compiling and Running a **terminfo** Program 9-57
An Example **terminfo** Program 9-57
Working with the **terminfo** Database 9-62
Writing Terminal Descriptions 9-62
Name the Terminal 9-62
Learn About the Capabilities 9-63
Specify Capabilities 9-64
Compile the Description 9-69
Test the Description 9-70
Comparing or Printing **terminfo** Descriptions 9-71
Converting a **termcap** Description to a

terminfo Description 9-72
curses Program Examples 9-73
The **editor** Program 9-73
The **highlight** Program 9-81
The **scatter** Program 9-83
The **show** Program 9-86
The **two** Program 9-88
The **window** Program 9-91

10. make

An Overview of the **make** Utility 10-1
Basic Features 10-2
Description Files and Substitutions 10-7
Comments 10-7
Continuation Lines 10-7
Macro Definitions 10-7
General Form 10-7
Dependency Information 10-8
Executable Commands 10-8
Extensions of $*, $@, and $< 10-9
Output Translations 10-9
The Recursive **Makefile** 10-11
Suffixes and Transformation Rules 10-11
Implicit Rules 10-11
Archive Libraries 10-13
SCCS Filenames: the Tilde 10-16
The Null Suffix 10-17
include Files 10-18
SCCS Makefiles 10-18
Dynamic Dependency Parameters 10-18
Command Usage 10-20
The **make** Command 10-20
Environment Variables 10-21
Suggestions and Warnings 10-23
Internal Rules 10-24

11. Source Code Control System (SCCS)

The Source Code Control System 11-1
SCCS for Beginners 11-2
Terminology 11-2

Creating an SCCS File via **admin** 11-2
Retrieving a File via **get** 11-3
Recording Changes via **delta** 11-4
Additional Information about **get** 11-4
The **help** Command 11-5
Delta Numbering 11-7
SCCS Command Conventions 11-10
x.files and z.files 11-10
Error Messages 11-11
SCCS Commands 11-12
The **get** Command 11-12
ID Keywords 11-13
Retrieval of Different Versions 11-14
Retrieval With Intent to Make a Delta 11-16
The **unget** Command 11-17
Additional **get** Options 11-17
Concurrent Edits of Different SID 11-18
Concurrent Edits of Same SID 11-21
Keyletters That Affect Output 11-21
The **delta** Command 11-23
The **admin** Command 11-25
Creation of SCCS Files 11-26
Inserting Commentary for the Initial Delta 11-26
Initialization and Modification of SCCS
File Parameters 11-27
The **prs** Command 11-28
The **sact** Command 11-30
The **help** Command 11-30
The **rmdel** Command 11-31
The **cdc** Command 11-31
The **what** Command 11-32
The **sccsdiff** Command 11-33
The **comb** Command 11-33
The **val** Command 11-34
The **vc** Command 11-35
SCCS Files 11-36
Protection 11-36
Formatting 11-37
Auditing 11-38

12. lint

The **lint** Program 12-1
Using **lint** 12-2
lint Message Types 12-4
Unused Variables and Functions 12-4
Set/Used Information 12-5
Flow of Control 12-5
Function Values 12-6
Type Checking 12-7
Type Casts 12-8
Nonportable Character Use 12-8
Assignments of **longs** to **ints** 12-9
Strange Construction 12-9
Old Syntax 12-10
Pointer Alignment 12-11
Multiple Uses and Side Effects 12-11

13. Shared Libraries

Introduction 13-1
Using a Shared Library 13-2
What Is a Shared Library? 13-2
The UNIX System Shared Libraries 13-3
Building an **a.out** File 13-3
Coding an Application 13-4
Deciding Whether to Use a Shared Library 13-5
More About Saving Space 13-6
Identifying **a.out** Files that Use Shared Libraries 13-12
Building a Shared Library 13-13
The Building Process 13-13
Step 1: Choosing Region Addresses 13-13
Step 2: Choosing the Target Library Pathname 13-14
Step 3: Selecting Library Contents 13-15
Step 4: Rewriting Existing Library Code 13-15
Step 5: Writing the Library Specification File 13-15
Step 6: Using **mkshlib** to Build the Host and Target 13-17

Guidelines for Writing Shared Library Code 13-18
Choosing Library Members 13-19
Changing Existing Code for the Shared Library 13-20
Importing Symbols 13-22
Providing Archive Library Compatibility 13-28
Tuning the Shared Library Code 13-29
Checking for Compatibility 13-30
Checking Versions of Shared Libraries 13-30
An Example 13-31
The Original Source 13-32
Choosing Region Addresses and the Target Pathname 13-36
Selecting Library Contents 13-36
Rewriting Existing Code 13-37
Writing the Specification File 13-38
Building the Shared Library 13-40
Summary 13-40

14. Using Real-Time Programming
Introduction 14-1
Real-Time Features 14-2
Interval Timers 14-2
Virtual Memory Control 14-3
Non-Degrading Priorities 14-4
Shared Resource Groups 14-5
Process Blocking 14-6
User Synchronization Primitives 14-6
Preemptable Kernel 14-6
Optimal Real-Time Environment 14-8
Establishing Multiprocessor Control 14-8
Locking Interrupts 14-8
A Real-Time Example 14-9
Real-Time Latency 14-16
The Components of Process Dispatch Latency 14-16
Maximum Process Dispatch Latency 14-17
Summary 14-18

A. Index to Utilities

Glossary

Purpose

This guide is designed to give you information about programming in a UNIX system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming languages by concentrating on the other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, we are addressing programmers, especially those who have not worked extensively with the UNIX system. No special level of programming involvement is assumed. We hope the book will be useful to people who write only an occasional program as well as those who work on or manage large application development projects.

Programmers in the expert class, or those engaged in developing system software, may find this guide lacks the depth of information they need. For them we recommend the *IRIS-4D Programmer's Reference Manual.*

Knowledge of terminal use, of a UNIX system editor, and of the UNIX system directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you might want to look over the *IRIS-4D User's Guide* before tackling this one.

Organization

The material is organized into two parts and twelve chapters, as follows:

- Part 1, Chapter 1 — Overview

  Identifies the special features of the UNIX system that make up the programming environment: the concept of building blocks, pipes, special files, shell programming, etc. As a framework for the material that follows, three different levels of programming in a UNIX system are defined: single-user, applications, and systems programming.

- Chapter 2 — Programming Basics

  Describes the most fundamental utilities needed to get programs running.

- Chapter 3 — Application Programming

  Enlarges on many of the topics covered in the previous chapter with particular emphasis on how things change as the project grows bigger. Describes tools for keeping programming projects organized.

- Part 2, Chapters 4 through 12 — Support Tools, Descriptions, and Tutorials

  Includes detailed information about the use of many of the UNIX system tools.

At the end of the text is a glossary and an index to the UNIX utilities.

The C Connection

The UNIX system supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the UNIX operating system and C has always been and remains very close. Most of the code in the UNIX operating system is C, and over the years many organizations using the UNIX system have come to use C for an increasing portion of their application code. Thus, while this guide is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C.

Hardware/Software Dependencies

The text reflects the way things work running UNIX System V at the Release 3.0 level. If you find commands that work a little differently in your UNIX system environment, it may be because you are running under a different release of the software. If some commands just don't seem to exist at all, they may be members of packages not installed on your system.

Notation Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout UNIX system documentation:

- Commands that you type in from your terminal are shown in **bold** type.
- Text that is printed on your terminal by the computer is shown in `typewriter` font. The typewriter font is also used for code samples because it allows the most accurate representation of spacing. Spacing is often a matter of coding style, but is sometimes critical.

- Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display.

  Italics are also used to show substitutable values, such as, *filename*, when the format of a command is shown.

- There is an implied return at the end of each command and menu response you enter. Where you may be expected to enter only a return (as in the case where you are accepting a menu default), the symbol **<return>** is used.

- In cases where you are expected to enter a control character, it is shown as, for example, **ctrl-d**. This means that you press the **d** key on your keyboard while holding down the control key.

- The percent sign, `%`, and pound sign, #, symbols are the standard default prompt signs for an ordinary user and **root** respectively. `%` means you are logged in as an ordinary user. # means you are logged in as **root**.

- When the # prompt is used in an example, it means the command illustrated may be used only by **root**.

Command References

When commands are mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses: **command**(section). The numbered sections are located in the following manuals:

Section (1)	*IRIS-4D User's Reference Manual*
Sections (1, 1M), (7), (8)	*IRIS-4D System Administrator's Reference Manual*
Sections (1), (2), (3), (4), (5)	*IRIS-4D Programmer's Reference Manual*

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the UNIX system software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, while we can't say that they have been compiled, we have attempted to maintain the same standards of coding accuracy for them.

The UNIX System Environment

The 1983 Turing Award of the Association for Computing Machinery was given jointly to Ken Thompson and Dennis Ritchie, the two men who first designed and developed the UNIX operating system. The award citation said, in part:

> "The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX system has led a generation of software designers to new ways of thinking about programming. The genius of the UNIX system is its framework which enables programmers to stand on the work of others."

As programmers working in a UNIX system environment, why should we care what Thompson and Ritchie did? Does it have any relevance for us today?

It does, because understanding the thinking behind the system design and the atmosphere in which it flowered can help us become productive UNIX system programmers faster.

The UNIX Idea

You may already have read about how Ken Thompson came across a DEC PDP-7 machine sitting unused in a hallway at AT&T Bell Laboratories, and how he and Dennis Ritchie and a few of their colleagues used that as the original machine for developing a new operating system that became UNIX.

The important thing to realize, however, is that what they were trying to do was fashion a pleasant computing environment for themselves. It was not, "Let's get together and build an operating system that will attract world-wide attention."

The sequence in which elements of the system fell into place is interesting. The first piece was the file system, followed quickly by its organization into a hierarchy of directories and files. The view of everything, data stores, programs, commands, directories, even devices, as files of one type or another was critical, as was the idea of a file as a one-dimensional array of bytes with no other structure implied. The simplicity of this way of looking at files has been a major contributing factor to a computer environment that programmers and other users have found comfortable to work in.

The next element was the idea of processes, with one process being able to create another and communicate with it. This innovative way of looking at running programs as processes led to the quintessentially UNIX practice of reusing code by calling it from another process. With the addition of commands to manipulate files

and an assembler to produce executable programs, the system was essentially able to function on its own.

The next major development was the acquisition of a DEC PDP-11 and the installation of the new system on it. This has been described by Ritchie as a stroke of good luck, in that the PDP-11 was to become a hugely successful machine, its success to some extent adding momentum to the acceptance of the system that began to be known by the name of UNIX.

By 1972 the innovative idea of pipes (connecting links between processes whereby the output of one becomes the input of the next) had been incorporated into the system, the operating system had been recoded in higher level languages (first B, then C), and had been dubbed with the name UNIX. By this point, the "pleasant computing environment" sought by Thompson and Ritchie was a reality; but some other things were going on that had a strong influence on the character of the product then and today.

It is worth pointing out that the UNIX system came out of an atmosphere that was totally different from that in which most commercially successful operating systems are produced. The more typical atmosphere is that described by Tracy Kidder in *The Soul of a New Machine*. In that case, dozens of talented programmers worked at white heat, in an atmosphere of extremely tight security, against murderous deadlines. By contrast, the UNIX system could be said to have had about a 10-year gestation period. From the beginning it attracted the interest of a growing number of brilliant specialists, many of whom found in the UNIX system an environment that allowed them to pursue research and development interests of their own, but who in turn contributed to the tools available for succeeding ranks of UNIX programmers.

Beginning in 1971, the system began to be used for applications within AT&T Bell Laboratories, and in 1974 was made available at low cost and without support to colleges and universities. These versions, called research versions and identified with Arabic numbers up through 7, occasionally grew on their own and fed additional innovative tools back to the main system. The widely used screen editor vi(1), for example, was added to the UNIX system by William Joy at the University of California, Berkeley.

Versions of the UNIX system being offered now are coming from an environment perhaps more closely related to the standard software factory. Features are being added to new releases in response to the expressed needs of the marketplace. The essential quality of the UNIX system, however, remains as the product of the innovative thinking of its originators and the cooperative atmosphere in which they worked. This quality has on occasion been referred to as the UNIX philosophy, but what is meant is the way in which sophisticated programmers have come to work with the UNIX system.

The UNIX Philosophy

For as long as you are writing programs on a UNIX system you should keep this motto hanging on your wall:

Build on the work of others

Unlike computer environments where each new project is like starting with a blank canvas, on a UNIX system a good percentage of any programming effort is lying there in **bins**, and **lbins**, and **/usr/bins**, not to mention **etc**, waiting to be used.

The features of the UNIX system (pipes, processes, and the file system) contribute to this reusability, as does the history of sharing and contributing that extends back to 1969. You risk missing the essential nature of the UNIX system if you don't put this to work.

UNIX System Tools

The term "UNIX system tools" can stand some clarification. In the narrowest sense, it means an existing piece of software used as a component in a new task. In a broader context, the term is often used to refer to elements of the UNIX system that might also be called features, utilities, programs, filters, commands, languages, functions, and so on. It gets confusing because any of the things that might be called by one or more of these names can be, and often are, used in the narrow way as part of the solution to a programming problem.

Tools Covered and Not Covered in This Guide

The *IRIS-4D Programmer's Guide* is about tools used in the process of creating programs in a UNIX system environment, so let's take a minute to talk about which tools we mean, which ones are not going to be covered in this book, and where you might find information about those not covered here.

Tools not covered in this text:

- the **login** procedure
- UNIX system editors and how to use them
- how the file system is organized and how you move around in it
- shell programming

Information about the preceding subjects can be found in the *IRIS-4D User's Guide* and in a number of commercially available texts.

Tools covered here can be classified as follows:

- utilities for getting programs running
- utilities for organizing software development projects
- specialized languages
- debugging and analysis tools
- compiled language components that are not part of the language syntax, for example, standard libraries, systems calls, and functions

The Shell as a Prototyping Tool

Any time you log in to a UNIX system machine you are using the shell. The shell is the interactive command interpreter that stands between you and the UNIX system kernel, but that's only part of the story. Because of its ability to start processes, direct the flow of control and field interrupts, and redirect input and output it is a full-fledged programming language. Programs that use these capabilities are known as shell procedures or shell scripts.

Innovative use of the shell involves stringing together commands to run under the control of a shell script. The commands that can be used in this way are documented in the *IRIS-4D User's Reference Manual.* Look through the *IRIS-4D User's Reference Manual* when you are trying to find a command with just the right option to handle a knotty programming problem. As you become familiar with the commands described in the manual pages you will be better able to take full advantage of the UNIX system environment.

It is not our purpose here to instruct you in shell programming. What we want to stress here is the important part that shell procedures can play in developing prototypes of full-scale applications. While understanding all the nuances of shell programming can be a fairly complex task, getting a shell procedure up and running is far less time-consuming than writing, compiling, and debugging compiled code.

This ability to get a program into production quickly is what makes the shell a valuable tool for program development. Shell programming allows you to "build on the work of others" to the greatest possible degree, since it allows you to piece together major components simply and efficiently. Many times even large applications can be done using shell procedures. Even if the application is initially developed as a prototype system for testing rather than for production purposes, many months of work can be saved.

With a prototype for testing, the range of possible user errors can be determined—something that is not always easy to plan for when an application is being designed. The method of dealing with strange user input can be worked out inexpensively, avoiding large re-coding problems.

A common occurrence in the UNIX system environment is to find that an available UNIX system tool can accomplish with a few lines of instructions what might take pages of compiled code. Shell procedures can intermix compiled modules and regular UNIX system commands to let you take advantage of work that has gone before.

Three Programming Environments

We distinguish among three programming environments because the information needs and the way in which UNIX system tools are used differ from one environment to another. We do not intend to imply a hierarchy of skill or experience. Highly skilled programmers can be found in the "single-user" category, and relative newcomers can be members of an application development or systems programming team.

Single-User Programming

Programmers in this environment are writing programs to perform their primary job. The resulting programs might well be added to the stock of programs available to the community in which the programmer works. This is similar to the atmosphere in which the UNIX system thrived; someone develops a useful tool and shares it with the rest of the organization. Single-user programmers may not have externally imposed requirements, or co-authors, or project management concerns. The programming task itself drives the coding very directly. One advantage of a timesharing system such as UNIX is that people with programming skills can be set free to work on their own without having to go through formal project approval channels and perhaps wait for months for a programming department to solve their problems.

Single-user programmers need to know how to:

- select an appropriate language
- compile and run programs
- use system libraries
- analyze programs
- debug programs
- keep track of program versions

Most of the information to perform these functions at the single-user level can be found in Chapter 2.

Application Programming

Programmers working in this environment are developing systems for the benefit of other, non-programming users. Most large commercial computer applications still involve a team of applications development programmers. They may be employees of the end-user organization or they may work for a software development firm. Some of the people working in this environment may be more in the project management area than working programmers.

Information needs of people in this environment include all the topics in Chapter 2, plus additional information on:

- software control systems
- file and record locking
- communication between processes
- shared memory
- advanced debugging techniques

These topics are discussed in Chapter 3.

Systems Programming

System programmers are writing software tools that are part of, or closely related to, the operating system itself. The project may involve writing a new device driver, a database management system or an enhancement to the UNIX system kernel. In addition to knowing their way around the operating system source code and how to make changes and enhancements to it, they need to be thoroughly familiar with all the topics covered in Chapters 2 and 3.

Summary

In this overview chapter we have described the way that the UNIX system developed and the effect that has on the way programmers now work with it. We have described what is and is not to be found in the other chapters of this guide to help programmers. We have also suggested that in many cases programming problems may be easily solved by taking advantage of the UNIX system interactive command interpreter known as the shell. Finally, we identified three programming environments in the hope that it will help orient the reader to the organization of the text in the remaining chapters.

Programming in a UNIX Environment

The information in this chapter is for anyone learning to write programs to run in a UNIX system environment. In Chapter 1 we identified one group of UNIX system users as single-user programmers. People in that category, particularly those who are not deeply interested in programming, may find this chapter (plus related reference manuals) tells them as much as they need to know about coding and running programs on a UNIX system computer.

Programmers whose interest does run deeper, who are part of an application development project, or who are producing programs on one UNIX system computer that are being ported to another, should view this chapter as a starter package.

Choosing a Programming Language

How do you decide which programming language to use in a given situation? One answer could be, "I always code in HAIRBOL, because that's the language I know best." Actually, in some circumstances that's a legitimate answer. But assuming more than one programming language is available to you, that different programming languages have their strengths and weaknesses, and that once you've learned to use one programming language it becomes relatively easy to learn to use another, you might approach the problem of language selection by asking yourself questions like the following:

- What is the nature of the task this program is to do?

  Does the task call for the development of a complex algorithm, or is this a simple procedure that has to be done on a lot of records?

- Does the programming task have many separate parts?

  Can the program be subdivided into separately compilable functions, or is it one module?

- How soon does the program have to be available?

  Is it needed right now, or do I have enough time to work out the most efficient process possible?

- What is the scope of its use?

  Am I the only person who will use this program, or is it going to be distributed to the whole world?

- Is there a possibility the program will be ported to other systems?
- What is the life-expectancy of the program?

  Is it going to be used just a few times, or will it still be going strong five years from now?

Supported Languages in the UNIX Environment

By "supported languages" we mean those running UNIX System V Release 3.0. Not all languages will necessarily be installed on your machine as they are separately sold items. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion. In this section and in the one to follow we give brief descriptions of the nature of four

full-scale programming languages, and a number of special-purpose languages.

C Language

C is intimately associated with the UNIX system since it was originally developed for use in recoding the UNIX system kernel. If you need to use a lot of UNIX system function calls for low-level I/O, memory or device management, or inter-process communication, C language is a logical first choice. Most programs, however, don't require such direct interfaces with the operating system so the decision to choose C might better be based on one or more of the following characteristics:

- a variety of data types: character, integer, long integer, float, and double
- low-level constructs (most of the UNIX system kernel is written in C)
- derived data types such as arrays, functions, pointers, structures, and unions
- multi-dimensional arrays
- scaled pointers, and the ability to do pointer arithmetic
- bit-wise operators
- a variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- a high degree of portability

C is a language that lends itself readily to structured programming. It is natural in C to think in terms of functions. The next logical step is to view each function as a separately compilable unit. This approach (coding a program in small pieces) eases the job of making changes and/or improvements. If this begins to sound like the UNIX system philosophy of building new programs from existing tools, it's not just coincidence. As you create functions for one program you will surely find that many can be picked up, or quickly revised, for another program.

A difficulty with C is that it takes a fairly concentrated use of the language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might make life easier if you choose a less-demanding language.

FORTRAN

The oldest of the high-level programming languages, FORTRAN is still highly prized for its variety of mathematical functions. If you are writing a program for statistical analysis or other scientific applications, FORTRAN is a good choice. An original design objective was to produce a language with good operating efficiency. This has been achieved at the expense of some flexibility in the area of type definition and data abstraction. There is, for example, only a single form of the iteration statement. FORTRAN also requires using a somewhat rigid format for input of lines of source code. This shortcoming may be overcome by using one of the UNIX system tools designed to make FORTRAN more flexible.

Assembly Language

The closest approach to machine language, assembly language is specific to the particular computer on which your program is to run. High-level languages are translated into the assembly language for a specific processor as one step of the compilation. The most common need to work in assembly language arises when you want to do some task that is not within the scope of a high-level language.

NOTE: Since assembly language is machine specific, programs written in it are not portable.

Special-Purpose Languages

In addition to the above formal programming languages, the UNIX system environment frequently offers one or more of the special-purpose languages listed below.

NOTE: Since UNIX system utilities and commands are packaged in functional groupings, it is possible that not all the facilities mentioned will be available on all systems.

awk

awk (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. On finding a line that matches a pattern, **awk** performs actions also described in the specification. It is not uncommon that an **awk** program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a

quantity. You have sorted the records by the key field, and you now want to add the quantities for records with duplicate keys and output a file in which no keys are duplicated. The pseudo-code for such a program might look like this:

```
Read the first record into a hold area;
Read additional records until EOF;
 {
 If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
 If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
 }
At EOF, write out the last record from the hold area.
```

An **awk** program to accomplish this task would look like this:

```
        { qty[$1] += $2 }
END     { for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**; its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudo-program.

lex

lex is a lexical analyzer that can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized, and pass the output stream on to the next program.

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammar rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammar rules.

M4

M4 is a macro processor that can be used as a preprocessor for assembly language and C programs. It is described in Section (1) of the *Programmer's Reference Manual.*

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**. You can use **dc** directly, if you want, but it takes a little getting used to since it works with reverse Polish notation. That means you enter numbers into a stack followed by the operator. **bc** and **dc** are described in Section (1) of the *IRIS-4D User's Reference Manual.*

curses

Actually a library of C functions, **curses** is included in this list because this set of functions almost amounts to a sub-language for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

In addition to all the preceding, don't overlook the possibility of using shell procedures.

Compiling and Link Editing

The command used for compiling depends on the language used;

- for C programs, **cc** both compiles and link edits
- for FORTRAN programs, **f77** both compiles and link edits

Compiling C Programs

To use the C compilation system you must have your source code in a file with a file name that ends in the characters **.c**, as in **mycode.c**. The command to invoke the compiler is:

```
cc mycode.c
```

If the compilation is successful the process proceeds through the link edit stage and the result will be an executable file by the name of **a.out**.

Several options to the **cc** command are available to control its operation. For a list of supported options, see the *IRIS-4D Series Compiler Guide.*

Compiling FORTAN Programs

The **f77** command invokes the FORTRAN compilation system. The operation of the command is similar to that of the **cc** command, except the source code file(s) must have a **.f** suffix. The **f77** command compiles your source code and calls in the link editor to produce an executable file whose name is **a.out**.

If you have the FORTRAN compiler a list of supported options can be found in *Porting FORTRAN Code to IRIS-4D Series Workstations.*

Compiler Diagnostic Messages

The C compiler generates error messages for statements that don't compile. The messages are generally understandable, but like most language compilers they sometimes point several statements beyond where the actual error occurred. For example, if you inadvertently put an extra ; at the end of an if statement, a subsequent else will be flagged as a syntax error. In the case where a block of several statements follows the if, the line number of the syntax error caused by the else will start you looking for the error well past where it is. Unbalanced curly braces, { }, are another common producer of syntax errors for which the error report may be delayed.

Link Editing

The **ld** command invokes the link editor directly. The typical user, however, seldom invokes **ld** directly. A more common practice is to use a language compilation control command (such as **cc**) that invokes **ld**. The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates start-up routines, and supports symbol table information used by **dbx**. You may, of course, start with a single object file rather than several. Unless an output filename is specified to **ld**, the resulting executable module is left in a file named **a.out**.

Any file named on the **ld** command line that is not an object file (typically, a name ending in **o**) is assumed to be an archive library or a file of link editor directives. The **ld** command has some 16 options. For a complete explanation of the link editor, see **ld**(1) in the *IRIS-4D Programmer's Reference Manual*, and the *IRIS-4D Series Compiler Guide.*

The UNIX System/Language Interface

A program depends on the operating system for a variety of services. Some of the services, such as bringing the program into main memory and starting the execution, are completely transparent. They are, in effect, arranged for in advance by the link editor when it marks an object module as executable. As a programmer you seldom need to be concerned about such matters.

Other services, however, such as input/output, file management, and storage allocation do require work on the part of the programmer. These connections between a program and the UNIX operating system are what is meant by the term UNIX system/language interface. The topics included in this section are:

- How arguments are passed to a program
- System calls and subroutines
- Header files and libraries
- Input/Output
- Processes
- Error Handling, Signals, and Interrupts

Using C to Illustrate the Interface

Throughout this section C programs are used to illustrate the interface between the UNIX system and programming languages because C programs make more use of the interface mechanisms than other high-level languages. What is really being covered in this section then is the UNIX system/C Language interface. The way that other languages deal with these topics is described in the user's guides for those languages.

Passing Arguments to a Program

Information or control data can be passed to a C program as arguments on the command line. When the program is run as a command, arguments on the command line are made available to the function **main** in two parameters, an argument count and an array of pointers to character strings. (Every C program is required to have an entry module by the name of **main.**) Since the argument count is always given, the program does not have to know in advance how many arguments to expect. The character strings pointed at by elements of the array of pointers contain the argument information.

The arguments are presented to the program traditionally as **argc** and **argv**, although any names you choose will work. **argc** is an integer that gives the count of the number of arguments. Since the command itself is considered to be the first argument, **argv[0]**, the count is always at least one. **argv** is an array of pointers to character strings (arrays of characters terminated by the null character \0).

If you plan to pass runtime parameters to your program, you need to include code to deal with the information. Two possible uses of runtime parameters are:

- as control data. Use the information to set internal flags that control the operation of the program.
- to provide a variable file name to the program.

Figures 2-1 and 2-2 show program fragments that illustrate these uses.

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
        void exit();
        int oflag = FALSE;
        int pflag = FALSE;          /* Function Flags */
        int rflag = FALSE;
        int ch;

                while ((ch = getopt(argc,argv, "opr")) != EOF)
                {
                  /* For options present, set flag to TRUE */
                  /* If no options present, print error message */
                        switch (ch)
                        {
                        case 'o':
                                oflag = 1;
                                break;
                        case 'p':
                                pflag = 1;
                                break;
                        case 'r':
                                rflag = 1;
                                break;
                        default:
                                (void)fprintf(stderr,
                                "Usage: %s [-opr]\n", argv[0]);
                                exit(2);
                        }
                }
                .
                .
                .
}
```

Figure 2-1: Using Command Line Arguments to Set Flags

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
        FILE *fopen(), *fin;
        void perror(), exit();

        if (argc > 1)
        {
                if ((fin = fopen(argv[1], "r")) == NULL)
                {
                  /* First string (%s) is program name (argv[0]) */
                  /* Second string (%s) is name of file that could */
                  /* not be opened (argv[1]) */

                         (void)fprintf(stderr,
                           "%s: cannot open %s: ",
                           argv[0], argv[1]);
                         perror("");
                         exit(2);
                }
        }
        .
        .
        .
}
```

Figure 2-2: Using **argv**[*n*] Pointers to Pass a Filename

The shell, which makes arguments available to your program, considers an argument to be any non-blank characters separated by blanks or tabs. Characters enclosed in double quotes ("abc def") are passed to the program as one argument even if blanks or tabs are among the characters.

A third argument is also present, in addition to **argc** and **argv**. The third argument, known as **envp**, is an array of pointers to environment variables. You can find more information on **envp** in the *IRIS-4D Programmer's Reference Manual* under **exec**(2) and **environ**(5).

System Calls and Subroutines

System calls are requests from a program for an action to be performed by the UNIX system kernel. Subroutines are precoded modules used to supplement the functionality of a programming language.

Both system calls and subroutines look like functions such as those you might code for the individual parts of your program. There are, however, differences between them:

- At link edit time, the code for subroutines is copied into the object file for your program; the code invoked by a system call remains in the kernel.
- At execution time, subroutine code is executed as if it was code you had written yourself; a system function call is executed by switching from your process area to the kernel.

This means that while subroutines make your executable object file larger, run-time overhead for context switching may be less and execution may be faster.

Categories of System Calls and Subroutines

System calls divide fairly neatly into the following categories:

- file access
- file and directory manipulation
- process control
- environment control and status information

You can generally tell the category of a subroutine by the section of the *IRIS-4D Programmer's Reference Manual* in which you find its manual page. However, the first part of Section 3 (3C and 3S) covers such a variety of subroutines it might be helpful to classify them further.

- The subroutines of sub-class 3S constitute the UNIX system/C Language standard I/O, an efficient I/O buffering scheme for C.
- The subroutines of sub-class 3C do a variety of tasks. They have in common the fact that their object code is stored in **libc.a**. They can be divided into the following categories:
  - string manipulation

- character conversion
- character classification
- environment management
- memory management

Figure 2-3 lists the functions that compose the standard I/O subroutines. Frequently, one manual page describes several related functions. In Figure 2-3 the left hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions described on the same manual page.

Function Name(s)				Purpose
fclose	fflush			close or flush a stream
ferror	feof	clearerr	fileno	stream status inquiries
fopen	freopen	fdopen		open a stream
fread	fwrite			binary input/output
fseek	rewind	ftell		reposition a file pointer in a stream
getc	getchar	fgetc	getw	get a character or word from a stream
gets	fgets			get a string from a stream
popen	pclose			begin or end a pipe to/from a process
printf	fprintf	sprintf		print formatted output

For all functions: #include <stdio.h>

The function name shown in **bold** gives the location in the *IRIS-4D Programmer's Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 1 of 2)

Function Name(s)				Purpose
putc	putchar	fputc	putw	put a character or word on a stream
puts	fputs			put a string on a stream
scanf	fscanf	sscanf		convert formatted input
setbuf	setvbuf			assign buffering to a stream
system				issue a command through the shell
tmpfile				create a temporary file
tmpnam	tempnam			create a name for a temporary file
ungetc				push character back into input stream
vprintf	vfprintf	vsprintf		print formatted output of a varargs argument list

For all functions: #include <stdio.h>

The function name shown in **bold** gives the location in the *IRIS-4D Programmer's Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 2 of 2)

Figure 2-4 lists string handling functions that are grouped under the heading **string**(3C) in the *IRIS-4D Programmer's Reference Manual.*

String Operations

strcat(s1, s2)	append a copy of s2 to the end of s1.
strncat(s1, s2, n)	append n characters from s2 to the end of s1.
strcmp(s1, s2)	compare two strings. Returns an integer less than, greater than or equal to 0 to show that s1 is lexico-graphically less than, greater than or equal to s2.
strncmp(s1, s2, n)	compare n characters from the two strings. Results are otherwise identical to strcmp.
strcpy(s1, s2)	copy s2 to s1, stopping after the null character (\0) has been copied.
strncpy(s1, s2, n)	copy n characters from s2 to s1. s2 will be truncated if it is longer than n, or padded with null characters if it is shorter than n.
strdup(s)	returns a pointer to a new string that is a duplicate of the string pointed to by s.
strchr(s, c)	returns a pointer to the first occurrence of character c in string s, or a NULL pointer if c is not in s.
strrchr(s, c)	returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c is not in s.

For all functions: #include <string.h>
string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 1 of 2)

String Operations

strlen(s)	returns the number of characters in s up to the first null character.
strpbrk(s1, s2)	returns a pointer to the first occurrence in s1 of any character from s2, or a NULL pointer if no character from s2 occurs in s1.
strspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters from s2.
strcspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters not from s2.
strtok(s1, s2)	look for occurrences of s2 within s1.

For all functions: #include <string.h>
string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 2 of 2)

Figure 2-5 lists macros that classify ASCII character-coded integer values. These macros are described under the heading **ctype**(3C) in Section 3 of the *IRIS-4D Programmer's Reference Manual.*

Classify Characters

isalpha(c)	is *c* a letter
isupper(c)	is *c* an uppercase letter
islower(c)	is *c* a lowercase letter
isdigit(c)	is *c* a digit [0-9]
isxdigit(c)	is *c* a hexadecimal digit [0-9], [A-F] or [a-f]
isalnum(c)	is *c* an alphanumeric (letter or digit)
isspace(c)	is *c* a space, tab, carriage return, new-line, vertical tab or form-feed
ispunct(c)	is *c* a punctuation character (neither control nor alphanumeric)
isprint(c)	is *c* a printing character, code 040 (space) through 0176 (tilde)
isgraph(c)	same as isprint except false for 040 (space)
iscntrl(c)	is *c* a control character (less than 040) or a delete character (0177)
isascii(c)	is *c* an ASCII character (code less than 0200)

For all functions: #include <ctype.h>
Nonzero return == true; zero return == false

Figure 2-5: Classifying ASCII Character-Coded Integer Values

Figure 2-6 lists functions and macros that are used to convert characters, integers, or strings from one representation to another.

Function Name(s)			Purpose
a64l	**l64a**		convert between long integer and base-64 ASCII string
ecvt	**fcvt**	**gcvt**	convert floating-point number to string
l3tol	**ltol3**		convert between 3-byte integer and long integer
strtod	**atof**		convert string to double-precision number
strtol	**atol**	**atoi**	convert string to integer

conv(3C):	**Translate Characters**
toupper	lowercase to uppercase
_toupper	macro version of toupper
tolower	uppercase to lowercase
_tolower	macro version of tolower
toascii	turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems

For all **conv**(3C) macros: #include <ctype.h>

Figure 2-6: Conversion Functions and Macros

Where the Manual Pages Can Be Found

System calls are listed alphabetically in Section 2 of the *IRIS-4D Programmer's Reference Manual*. Subroutines are listed in Section 3. We have described above what is in the first subsection of Section 3. The remaining subsections of Section 3 are:

- 3M—functions that make up the Math Library, **libm**
- 3X—various specialized functions
- 3F—the FORTRAN intrinsic function library, **libF77**
- 3N—networking functions
- 3—functions used by multiple languages

Using System Calls and Subroutines in C Programs

Information about the proper way to use system calls and subroutines is given on the manual page, but you have to know what you are looking for before it begins to make sense. To illustrate, a typical manual page (for **gets**(3S)) is shown in Figure 2-7.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char   *gets (s)
char   *s;

char   *fgets (s, n, stream)
char   *s;
int   n;
FILE    *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S),
fopen(3S),
fread(3S),
getc(3S),
scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

Figure 2-7: Manual Page for **gets(3S)**

As you can see from the illustration, two related functions are described on this page: **gets** and **fgets**. Each function gets a string from a stream in a slightly different way. The DESCRIPTION section tells how each operates.

It is the SYNOPSIS section, however, that contains the critical information about how the function (or macro) is used in your program. Notice in Figure 2-7 that the first line in the SYNOPSIS is

#include <stdio.h>

This means that to use **gets** or **fgets** you must bring the standard I/O header file into your program (generally right at the top of the file). There is something in **stdio.h** that is needed when you use the described functions. Figure 2-9 shows a version of **stdio.h**. Check it to see if you can understand what **gets** or **fgets** uses.

The next thing shown in the SYNOPSIS section of a manual page that documents system calls or subroutines is the formal declaration of the function. The formal declaration tells you:

- the type of object returned by the function

  In our example, both **gets** and **fgets** return a character pointer.

- the object or objects the function expects to receive when called

  These are the things enclosed in the parentheses of the function. **gets** expects a character pointer. (The DESCRIPTION section sheds light on what the tokens of the formal declaration stand for.)

- how the function is going to treat those objects

  The declaration

```
char *s;
```

in **gets** means that the token s enclosed in the parentheses will be considered to be a pointer to a character string. Bear in mind that in the C language, when passed as an argument, the name of an array is converted to a pointer to the beginning of the array.

We have chosen a simple example here in **gets**. If you want to test yourself on something a little more complex, try working out the meaning of the elements of the **fgets** declaration.

While we're on the subject of **fgets**, there is another piece of C esoterica that we'll explain. Notice that the third parameter in the **fgets** declaration is referred to as **stream**. A **stream**, in this context, is a file with its associated buffering. It is declared to be a pointer to a defined type FILE. Where is FILE defined? Right! In **stdio.h**.

To finish off this discussion of the way you use functions described in the *IRIS-4D Programmer's Reference Manual* in your own code, in Figure 2-8 we show a program fragment in which **gets** is used.

```
#include <stdio.h>

main()
{
        char sarray[80];

        for(;;)
        {
                if (gets(sarray) != NULL)
                .
                .       /* Do something with the string */
                .
        }
}
```

Figure 2-8: How **gets** Is Used in a Program

You might ask, "Where is **gets** reading from?" The answer is, "From the standard input." This generally means from something being keyed in from the terminal or output from another command that was piped to **gets**. How do we know that? The DESCRIPTION section of the **gets** manual page says, "**gets** reads characters from the standard input...." Where is the standard input defined? In **stdio.h**.

```
#ifndef _NFILE
#define _NFILE 20

#define BUFSIZ 1024
#define _SBFSIZ 8

typedef struct {
                int             _cnt;
                unsigned char   *_ptr;
                unsigned char   *_base;
                char            _flag;
                char            _file;
} FILE;

#define _IOFBF                  0000  /* _IOLBF means that a file's output  */
#define _IOREAD                 0001  /* will be buffered line by line.     */
#define _IOWRT                  0002  /* In addition to being flags, _IONBF,*/
#define _IONBF                  0004  /* _IOLBF and IOFBF are possible      */
#define _IOMYBUF                0010  /* values for "type" in setvbuf.      */
#define _IOEOF                  0020
#define _IOERR                  0040
#define _IOLBF                  0100
#define _IORW                   0200

#ifndef NULL
#define NULL                    0
#endif
#ifndef EOF
#define EOF                     (-1)
#endif
```

Figure 2-9: A Version of **stdio.h** (sheet 1 of 2)

```
#define stdin             (&_iob[0])
#define stdout            (&_iob[1])
#define stderr            (&_iob[2])

#define _bufend(p)        _bufendtab[(p)->_file]
#define _bufsiz(p)        (_bufend(p) - (p)->_base)

#ifndef lint
#define getc(p)           (--(p)->_cnt <0 ? _filbuf(p):(int)*(p)->_ptr++)
#define putc(x, p)        (--(p)->_cnt < 0 ?
                          _flsbuf((unsigned char) (x), (p)) :
                          (int) (*(p)->_ptr++ = (unsigned char) (x)))
#define getchar()         getc(stdin)
#define putchar(x)        putc((x), stdout)
#define clearerr(p)       ((void) ((p)->_flag &= (_IOERR | _IOEOF)))
#define feof(p)           ((p)->_flag & _IOEOF)
#define ferror(p)         ((p)->_flag & _IOERR)
#define fileno(p)         (p)->_file
#endif

extern FILE     _iob[_NFILE];
extern FILE     *fopen(),*fdopen(),*freopen(),*popen(),*tmpfile();
extern long     ftell();
extern void     rewind(), setbuf();
extern char     *ctermid(),*cuserid(),*fgets(),*gets(),*tempnam(),*tmpnam();
extern unsigned char *_bufendtab[];

#define L_ctermid         9
#define L_cuserid         9
#define P_tmpdir          "/usr/tmp/"
#define L_tmpnam          (sizeof(P_tmpdir) + 15)
#endif
```

Figure 2-9: A Version of **stdio.h** (sheet 2 of 2)

Header Files and Libraries

In the earlier parts of this chapter there have been frequent references to **stdio.h**, and a version of the file itself is shown in Figure 2-9. **stdio.h** is the most commonly used header file in the UNIX system/C environment, but there are many others.

Header files carry definitions and declarations that are used by more than one function. Header file names traditionally have the suffix **.h**, and are brought into a program at compile time by the C preprocessor. The preprocessor does this because it interprets the **#include** statement in your program as a directive; as indeed it is. All keywords preceded by a pound sign (#) at the beginning of the line, are treated as preprocessor directives. The two most commonly used directives are **#include** and **#define**. We have already seen that the **#include** directive is used to call in (and process) the contents of the named file. The **#define** directive is used to replace a name with a token string. For example,

#define _NFILE 20

sets to 20 the number of files a program can have open at one time. See **cpp**(1) for the complete list.

In the pages of the *IRIS-4D Programmer's Reference Manual* there are about 45 different **.h** files named. The format of the **#include** statement for all these shows the file name enclosed in angle brackets (<>), as in

#include <stdio.h>

The angle brackets tell the C preprocessor to look in the standard places for the file. In most systems the standard place is in the **/usr/include** directory. If you have some definitions or external declarations that you want to make available in several files, you can create a **.h** file with any editor, store it in a convenient directory and make it the subject of a **#include** statement such as the following:

#include "../defs/rec.h"

It is necessary, in this case, to provide the relative pathname of the file and enclose it in quotation marks. Fully qualified pathnames (those that begin with /) can create portability and organizational problems. An alternative to long or fully qualified pathnames is to use the **-I***dir* preprocessor option when you compile the program. This option directs the preprocessor to search for **#include** files whose names are enclosed in quotation marks, first in the directory of the file being compiled, then in the directories named in the **-I** option(s), and finally in directories on

the standard list. In addition, all **#include** files whose names are enclosed in angle brackets (< >) are first searched for in the list of directories named in the **-I** option and finally in the directories on the standard list.

Object File Libraries

It is common practice in UNIX system computers to keep modules of compiled code (object files) in archives; by convention these are designated by a **.a** suffix. System calls from Section 2, and the subroutines that are functions (as distinct from macros) found in Section 3, subsections 3C and 3S of the *IRIS-4D Programmer's Reference Manual* are kept in an archive file by the name of **libc.a**. In most systems, **libc.a** is found in the directory **/lib**. Many systems also have a directory **/usr/lib**. Where both **/lib** and **/usr/lib** occur, **/usr/lib** is apt to be used to hold archives that are related to specific applications.

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default the **cc** command that invokes the C compilation system causes the link editor to search **libc.a**. If you need to point the link editor to other libraries that are not searched by default, you do it by naming them explicitly on the command line with the **-l** option. The format of the **-l** option is **-l***x* where *x* is the library name, and can be up to nine characters. For example, if your program includes functions from the **curses** screen control package, the option

-lcurses

will cause the link editor to search for **/lib/libcurses.a** or **/usr/lib/libcurses.a** and use the first one it finds to resolve references in your program.

In cases where you want to direct the order in which archive libraries are searched, you may use the **-L** *dir* option. Assuming the **-L** option appears on the command line ahead of the **-l** option, it directs the link editor to search the named directory for **lib***x***.a** before looking in **/lib** and **/usr/lib**. This is particularly useful if you are testing out a new version of a function that already exists in an archive in a standard directory. Its success is due to the fact that once having resolved a reference the link editor stops looking. That's why the **-L** option, if used, should appear on the command line ahead of any **-l** specification.

Input and Output

We talked some about I/O earlier in this chapter in connection with system calls and subroutines. A whole set of subroutines constitutes the C language standard I/O package, and there are several system calls that deal with the same area. In this section we want to get into the subject in a little more detail and describe for you how to deal with input and output concerns in your C programs. First off, let's briefly define what I/O encompasses. It has to do with

- creating and sometimes removing files
- opening and closing files used by your program
- transferring information from a file to your program (reading)
- transferring information from your program to a file (writing)

In this section we will describe some of the subroutines you might choose for transferring information, but the heaviest emphasis will be on dealing with files.

Three Files to Handle Standard I/O Streams

Programs are permitted to have several files open simultaneously. The number may vary from system to system; the most common maximum is 20. _NFILE in **stdio.h** specifies the number of standard I/O files a program is permitted to have open.

Any program automatically starts off with three files. If you will look again at Figure 2-9, about midway through you will see that **stdio.h** contains three **#define** directives that equate **stdin**, **stdout**, and **stderr** to the address of _iob[0], _iob[1], and _iob[2], respectively. The array _iob holds information dealing with the way standard I/O handles streams. It is a representation of the open file table in the control block for your program. The position in the array is a digit that is also known as the file descriptor. The default in UNIX systems is to associate all three of these files with your terminal.

The real significance is that functions and macros that deal with **stdin** or **stdout** can be used in your program with no further need to open or close files. For example, **gets**, cited above, reads a string from **stdin**; **puts** writes a null-terminated string to **stdout**. There are others that do the same thing in slightly different ways, for example, character at a time and formatted. You can specify that output be directed to **stderr** by using a function such as **fprintf**. **fprintf** works the same as **printf** except that it delivers its formatted output to a named stream, such as **stderr**. You can use the shell's redirection feature on the command line to read from or write into a named file. If you want to separate error messages from ordinary output being sent to **stdout** and thence possibly piped by the shell to a succeeding

program, you can do it by using one function to handle the ordinary output and a variation of the same function that names the stream, to handle error messages.

Named Files

Any files other than **stdin**, **stdout**, and **stderr** that are to be used by your program must be explicitly connected by you before the file can be read from or written to. This can be done using the standard library routine **fopen**. **fopen** takes a pathname (which is the name by which the file is known to the UNIX file system), asks the system to keep track of the connection, and returns a pointer that you then use in functions that do the reads and writes.

A structure is defined in **stdio.h** with a type of FILE. In your program you need to have a declaration such as

```
FILE *fin;
```

The declaration says that **fin** is a pointer to a FILE. You can then assign the name of a particular file to the pointer with a statement in your program like this:

```
fin = fopen("filename", "r");
```

where **filename** is the pathname to open. The "f3r" means that the file is to be opened for reading. This argument is known as the **mode**. As you might suspect, there are separate modes for reading, writing, and both reading and writing. Actually, the file open function is often included in an if statement such as:

```
if ((fin = fopen("filename", "r")) == NULL)
 (void)fprintf(stderr,"%s:Unable to open input file %s\n",argv[0],"filename");
```

that takes advantage of the fact that **fopen** returns a NULL pointer if it can't open the file.

Once the file has been successfully opened, the pointer **fin** is used in functions (or macros) to refer to the file. For example:

```
int c;
c = getc(fin);
```

brings in a character at a time from the file into an integer variable called **c**. The variable **c** is declared as an integer even though we are reading characters because the function **getc()** returns an integer. Getting a character is often incorporated into some flow-of-control mechanism such as:

```
while ((c = getc(fin)) != EOF)
        .
        .
        .
```

that reads through the file until EOF is returned. EOF, NULL, and the macro **getc** are all defined in **stdio.h. getc** and others that make up the standard I/O package keep advancing a pointer through the buffer associated with the file; the UNIX system and the standard I/O subroutines are responsible for seeing that the buffer is refilled (or written to the output file if you are producing output) when the pointer reaches the end of the buffer. All these mechanics are mercifully invisible to the program and the programmer.

The function **fclose** is used to break the connection between the pointer in your program and the pathname. The pointer may then be associated with another file by another call to **fopen**. This re-use of a file descriptor for a different stream may be necessary if your program has many files to open. For output files it is good to issue an **fclose** call because the call makes sure that all output has been sent from the output buffer before disconnecting the file. The system call **exit** closes all open files for you. It also gets you completely out of your process, however, so it is safe to use only when you are sure you are completely finished.

The UNIX System and Low-level I/O

The term low-level I/O refers to the process of using system calls from Section 2 of the *IRIS-4D Programmer's Reference Manual* rather than the functions and subroutines of the standard I/O package. We are going to postpone until Chapter 3 any discussion of when this might be advantageous. If you find as you go through the information in this chapter that it is a good fit with your programming objectives, it is a safe assumption that you can work with C language programs in the UNIX system for a good many years without ever having a real need to use system calls to handle your I/O and file accessing problems. The reason low-level I/O is perilous is because it is more system-dependent. Your programs are less portable and probably no more efficient.

System Calls for Environment or Status Information

Under some circumstances you might want to be able to monitor or control the environment in your computer. There are system calls that can be used for this purpose. Some of them are shown in Figure 2-10.

Function Name(s)		Purpose
chdir		change working directory
chmod		change access permission of a file
chown		change owner and group of a file
getpid	**getpgrp getppid**	get process IDs
getuid	**geteuid getgid**	get user IDs
ioctl		control device
link	**unlink**	add or remove a directory entry
mount	**umount**	mount or unmount a file system
nice		change priority of a process
stat	**fstat**	get file status
time		get time
ulimit		get and set user limits
uname		get name of current UNIX system

Figure 2-10: Environment and Status System Calls

Many of the functions shown in Figure 2-10 have equivalent UNIX system shell commands. Shell commands can easily be incorporated into shell scripts to accomplish the monitoring and control tasks you may need to do. The functions are available, however, and may be used in C programs as part of the UNIX system/C Language interface. They are documented in Section 2 of the *IRIS-4D Programmer's Reference Manual.*

Processes

Whenever you execute a command in the UNIX system you are initiating a process that is numbered and tracked by the operating system. A flexible feature of the UNIX system is that processes can be generated by other processes. This happens more than you might ever be aware of. For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as **vi**, take the option of invoking the shell from **vi**, and execute the **ps** command, you will see a display something like that in Figure 2-11 (which shows the results of a **ps -f** command):

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
abc	24210	1	0	06:13:14	tty29	0:05	-sh
abc	24631	24210	0	06:59:07	tty29	0:13	vi c2.uli
abc	28441	28358	80	09:17:22	tty29	0:01	ps -f
abc	28358	24631	2	09:15:14	tty29	0:01	sh -i

Figure 2-11: Process Status

As you can see, user abc (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is Process 24210; its parent is the initialization process (Process ID 1). Process 24210 is the parent of Process 24631, and so on.

The four processes in the example above are all UNIX system shell level commands, but you can spawn new processes from your own program. (Actually, when you issue the command from your terminal to execute a program you are asking the shell to start another process, the process being your executable object module with all the functions and subroutines that were made a part of it by the link editor.)

You might think, "Well, it's one thing to switch from one program to another when I'm at my terminal working interactively with the computer; but why would a program want to run other programs, and if one does, why wouldn't I just put everything together into one big executable module?"

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing. (If it's the end of the month, go do a trial balance, for example.) The usual reasons why it might not be practical to create one huge executable are:

- The load module may get too big to fit in the maximum process size for your system.
- You may not have control over the object code of all the other modules you want to include.

There are legitimate reasons why this creation of new processes might need to be done. There are three ways to do it:

- **system**(3S)—request the shell to execute a command
- **exec**(2)—stop this process and start another
- **fork**(2)—start an additional copy of this process

system(3S)

The formal declaration of the **system** function looks like this:

```
#include <stdio.h>

int system(string)
char *string;
```

The function asks the shell to treat the string as a command line. The string can therefore be the name and arguments of any executable program or UNIX system shell command. If the exact arguments vary from one execution to the next, you may want to use **sprintf** to format the string before issuing the **system** command. When the command has finished running, **system** returns the shell exit status to your program. Execution of your program waits for the completion of the command initiated by **system** and then picks up again at the next executable statement.

exec(2)

exec is the name of a family of functions that includes **execv**, **execle**, **execve**, **execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**execl**) might be:

```
execl("/bin/prog2", "prog", progarg1, progarg2, (char *)0);
```

For **execl** the argument list is

/bin/prog2	pathname of the new process file
prog	the name the new process gets in its argv[0]
progarg1, progarg2	arguments to *prog2* as char *'s
(char *)0	a null char pointer to mark the end of the arguments

Check the manual page in the *IRIS-4D Programmer's Reference Manual* for the rest of the details. The key point of the **exec** family is that there is no return from a successful execution: the calling process is finished, the new process overlays the old. The new process also takes over the Process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of -1. You can check **errno** (see below) to learn why it failed.

fork(2)

The **fork** system call creates a new process that is an exact copy of the calling process. The new process is known as the child process; the caller is known as the parent process. The one major difference between the two processes is that the child gets its own unique process ID. When the **fork** process has completed successfully, it returns a 0 to the child process and the child's process ID to the parent. If the idea of having two identical processes seems a little funny, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could say, "Okay, I'm the child. I'm supposed to issue an **exec** for an entirely different program."
- The parent process could say, "My child is going to be **execing** a new process. I'll issue a **wait** until I get word that that process is finished."

In the world of C programming language your code might include statements like this:

```
#include <errno.h>

int ch_stat, ch_pid, status;
char *progarg1;
char *progarg2;
void exit();
extern int errno;

     if ((ch_pid = fork()) < 0)
     {
          /* Could not fork...
             check errno
          */
     }
     else if (ch_pid == 0)                    /* child */
     {
          (void)execl("/bin/prog2","prog",progarg1,progarg2,(char *)0);
          exit(2);   /* execl() failed */
     }
     else            /* parent */
     {
          while ((status = wait(&ch_stat)) != ch_pid)
          {
               if (status < 0 && errno == ECHILD)
                 break;
               errno = 0;
          }
     }
```

Figure 2-12: Example of **fork**

Because the child process ID is taken over by the new **exec**'d process, the parent knows the ID. This is a way of leaving one program to run another, enabling the system to return to the point in the first program where processing left off. This is exactly what the **system**(3S) function does. **system** accomplishes it through this same procedure of **forking** and **execing**, with a **wait** in the parent.

Keep in mind that the fragment of code above includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are

automatically opened: **stdin**, **stdout**, and **stderr**. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Pipes

The idea of using pipes - a connection between the output of one program and the input of another - when working with commands executed by the shell is well established in the UNIX system environment. For example, to learn the number of archive files in your system you might enter a command like:

echo /lib/*.a /usr/lib/*.a | wc -w

that first echoes all the files in **/lib** and **/usr/lib** that end in **.a**, then pipes the results to the **wc** command, which counts their number.

A feature of the UNIX system/C Language interface is the ability to establish pipe connections between your process and a command to be executed by the shell, or between two cooperating processes. The first uses the **popen**(3S) subroutine that is part of the standard I/O package; the second requires the system call **pipe**(2).

popen is similar in concept to the **system** subroutine in that it causes the shell to execute a command. The difference is that once having invoked **popen** from your program, you have established an open line to a concurrently running process through a stream. You can send characters or strings to this stream with standard I/O subroutines just as you would to **stdout** or to a named file. The connection remains open until your program invokes the companion **pclose** subroutine. A common application of this technique might be a pipe to a printer spooler. For example:

```
#include <stdio.h>

main()
{
      FILE *pptr;
      char *outstring;

      if ((pptr = popen("lp","w")) != NULL)
      {
            for(;;)
            {     .
                  .     /* Organize output */
                  .
                  (void)fprintf(pptr, "%s\n", outstring);
                  .
                  .
                  .
            }
         .
         .
         .
         pclose(pptr);
          }
          .
          .
          .
}
```

Figure 2-13: Example of a **popen** pipe

Error Handling

Within your C programs you must determine the appropriate level of checking for valid data and for acceptable return codes from functions and subroutines. If you use any of the system calls described in Section 2 of the *IRIS-4D Programmer's Reference Manual*, you have a way in which you can find out the probable cause of a bad return value.

UNIX system calls that are not able to complete successfully almost always return a value of -1 to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined, but they are the exceptions.) In addition to the -1 that is returned to the program, the unsuccessful system call places an integer in an externally declared variable, **errno**. You can determine the value in **errno** if your program contains the statement

```
#include <errno.h>
```

The value in **errno** is not cleared on successful calls, so your program should check it only if the system call returned a -1. The errors are described in **intro**(2) of the *IRIS-4D Programmer's Reference Manual.*

The subroutine **perror**(3C) can be used to print an error message (on **stderr**) based on the value of **errno.**

Signals and Interrupts

Signals and interrupts are two words for the same thing. Both refer to messages passed by the UNIX system to running processes. Generally, the effect is to cause the process to stop running. Some signals are generated if the process attempts to do something illegal; others can be initiated by a user against his or her own processes, or by the super-user against any process.

There is a system call, **kill**, that you can include in your program to send signals to other processes running under your user-id. The format for the **kill** call is:

```
kill(pid, sig)
```

where **pid** is the process number against which the call is directed, and **sig** is an integer from 1 to 19 that shows the intent of the message. The name "kill" is something of an overstatement; not all the messages have a "drop dead" meaning. Some of the available signals are shown in Figure 2-14 as they are defined in **<sys/signal.h>**.

```
#define   SIGHUP     1    /* hangup */
#define   SIGINT     2    /* interrupt (rubout) */
#define   SIGQUIT    3    /* quit (ASCII FS) */
#define   SIGILL     4    /* illegal instruction (not reset when caught)*/
#define   SIGTRAP    5    /* trace trap (not reset when caught) */
#define   SIGIOT     6    /* IOT instruction */
#define   SIGABRT    6    /* used by abort, replace SIGIOT in the future */
#define   SIGEMT     7    /* EMT instruction */
#define   SIGFPE     8    /* floating point exception */
#define   SIGKILL    9    /* kill (cannot be caught or ignored) */
#define   SIGBUS     10   /* bus error */
#define   SIGSEGV    11   /* segmentation violation */
#define   SIGSYS     12   /* bad argument to system call */
#define   SIGPIPE    13   /* write on a pipe with no one to read it */
#define   SIGALRM    14   /* alarm clock */
#define   SIGTERM    15   /* software termination signal from kill */
#define   SIGUSR1    16   /* user defined signal 1 */
#define   SIGUSR2    17   /* user defined signal 2 */
#define   SIGCLD     18   /* death of a child */
#define   SIGPWR     19   /* power-fail restart */


#define   SIGPOLL 22  /* pollable event occurred */

#define   NSIG       23   /* The valid signal number is from 1 to NSIG-1 */
#define   MAXSIG     32   /* size of u_signal[], NSIG-1 <= MAXSIG*/
                          /* MAXSIG is larger than we need now. */
                          /* In the future, we can add more signal */
                          /* number without changing user.h */
```

Figure 2-14: Signal Numbers Defined in **/usr/include/sys/signal.h**

The **signal**(2) system call is designed to let you code methods of dealing with incoming signals. You have a three-way choice. You can a) accept whatever the default action is for the signal, b) have your program ignore the signal, or c) write a function of your own to deal with it.

Analysis and Debugging

The UNIX system provides several commands designed to help you discover what causes problems in programs and to learn about potential problems.

Sample Program

To illustrate how these commands are used and the type of output they produce, we have constructed a sample program that opens and reads an input file and performs one to three subroutines according to options specified on the command line. This program does not do anything you couldn't do on your pocket calculator, but it does serve to illustrate some points. The source code is shown in Figure 2-15. The header file, **recdef.h**, is shown at the end of the source code.

The output produced by the various analysis and debugging tools illustrated in this section may vary slightly from one installation to another. The *IRIS-4D Programmer's Reference Manual* is a good source of additional information about the contents of the reports.

```
/* Main module -- restate.c */

#include <stdio.h>
#include "recdef.h"

#define TRUE    1
#define FALSE   0

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fopen(), *fin;
     void exit();
     int getopt();
     int oflag = FALSE;
     int pflag = FALSE;
     int rflag = FALSE;
     int ch;
     struct rec first;
     extern int opterr;
     extern float oppty(), pft(), rfe();

           /* restate.c is continued on the next page */
```

Figure 2-15: Source Code for Sample Program (sheet 1 of 4)

```
                    /* restate.c continued */

    if (argc < 2)
    {
          (void) fprintf(stderr, "%s: Must specify option\n",argv[0]);
          (void) fprintf(stderr, "Usage: %s -rpo\n", argv[0]);
          exit(2);
    }

    opterr = FALSE;
    while ((ch = getopt(argc,argv,"opr")) != EOF)
    {
          switch(ch)
          {
          case 'o':
                oflag = TRUE;
                break;
          case 'p':
                pflag = TRUE;
                break;
          case 'r':
                rflag = TRUE;
                break;
          default:
                (void) fprintf(stderr, "Usage: %s -rpo\n",argv[0]);
                exit(2);
          }
    }
    if ((fin = fopen("info","r")) == NULL)
    {
    (void) fprintf(stderr, "%s: cannot open input file %s\n",argv[0],"info");
    exit(2);
    }
```

Figure 2-15: Source Code for Sample Program (sheet 2 of 4)

```
        /* restate.c continued */

if (fscanf(fin, "%s%f%f%f%f%f%f",first.pname,&first.ppx,
&first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
{
  (void) fprintf(stderr,"%s: cannot read first record from %s\n",
     argv[0],"info");
  exit(2);
}

printf("Property: %s\n",first.pname);

if(oflag)
     printf("     Opportunity Cost: $%#5.2f\n",oppty(&first));

if(pflag)
     printf("     Anticipated Profit(loss): $%#7.2f\n",pft(&first));

if(rflag)
     printf("     Return on Funds Employed: %#3.2f%%\n",rfe(&first));
}
        /* End of Main Module -- restate.c */

        /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
     return(ps->i/12 * ps->t * ps->dp);
}
```

Figure 2-15: Source Code for Sample Program (sheet 3 of 4)

```
                    /* Profit -- pft.c */

#include "recdef.h"

float
pft(ps)
struct rec *ps;
{
    return(ps->spx - ps->ppx + ps->c);
}

                    /* Return on Funds Employed -- rfe.c */

#include "recdef.h"

float
rfe(ps)
struct rec *ps;
{
    return(100 * (ps->spx - ps->c) / ps->spx);
}

                    /* Header File -- recdef.h */

struct rec {            /* To hold input  */
    char pname[25];
    float ppx;
    float dp;
    float i;
    float c;
    float t;
    float spx;
  };
```

Figure 2-15: Source Code for Sample Program (sheet 4 of 4)

cflow

cflow produces a chart of the external references in C, **yacc**, **lex**, and assembly language files. Using the modules of our sample program, the command

cflow restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-16.

```
1    main: int(), <restate.c 11>
2         fprintf: <>
3         exit: <>
4         getopt: <>
5         fopen: <>
6         fscanf: <>
7         printf: <>
8         oppty: float(), <oppty.c 7>
9         pft: float(), <pft.c 7>
10        rfe: float(), <rfe.c 8>
```

Figure 2-16: **cflow** Output, No Options

The **-r** option looks at the caller:callee relationship from the other side. It produces the output shown in Figure 2-17.

```
1    exit: <>
2         main : <>
3    fopen: <>
4         main : 2
5    fprintf: <>
6         main : 2
7    fscanf: <>
8         main : 2
9    getopt: <>
10        main : 2
11   main: int(), <restate.c 11>
12   oppty: float(), <oppty.c 7>
13        main : 2
14   pft: float(), <pft.c 7>
15        main : 2
16   printf: <>
17        main : 2
18   rfe: float(), <rfe.c 8>
19        main : 2
```

Figure 2-17: **cflow** Output, Using **-r** Option

The **-ix** option causes external and static data symbols to be included. Our sample program has only one such symbol, **opterr**. The output is shown in Figure 2-18.

```
1     main: int(), <restate.c 11>
2           fprintf: <>
3           exit: <>
4           opterr: <>
5           getopt: <>
6           fopen: <>
7           fscanf: <>
8           printf: <>
9           oppty: float(), <oppty.c 7>
10          pft: float(), <pft.c 7>
11          rfe: float(), <rfe.c 8>
```

Figure 2-18: **cflow** Output, Using **-ix** Option

Combining the **-r** and the **-ix** options produces the output shown in Figure 2-19.

```
1    exit: <>
2         main : <>
3    fopen: <>
4         main : 2
5    fprintf: <>
6         main : 2
7    fscanf: <>
8         main : 2
9    getopt: <>
10        main : 2
11   main: int(), <restate.c 11>
12   oppty: float(), <oppty.c 7>
13        main : 2
14   opterr: <>
15        main : 2
16   pft: float(), <pft.c 7>
17        main : 2
18   printf: <>
19        main : 2
20   rfe: float(), <rfe.c 8>
21        main : 2
```

Figure 2-19: **cflow** Output, Using **-r** and **-ix** Options

ctrace

ctrace lets you follow the execution of a C program statement by statement. **ctrace** takes a **.c** file as input and inserts statements in the source code to print out variables as each program statement is executed. You must direct the output of this process to a temporary **.c** file. The temporary file is then used as input to **cc**. When the resulting **a.out** file is executed it produces output that can tell you a lot about what is going on in your program.

Options let you limit the number of times through loops. You can also include functions in your source file that turn the trace off and on so you can limit the output to portions of the program that are of particular interest.

ctrace accepts only one source code file as input. To use our sample program to illustrate, it is necessary to execute the following four commands:

```
ctrace restate.c > ct.main.c
ctrace oppty.c > ct.op.c
ctrace pft.c > ct.p.c
ctrace rfe.c > ct.r.c
```

The names of the output files are completely arbitrary. Use any names that are convenient for you. The names must end in **.c**, since the files are used as input to the C compilation system.

```
cc -o ct.run ct.main.c ct.op.c ct.p.c ct.r.c
```

Now the command

```
ct.run -opr
```

produces the output shown in Figure 2-20. The command above will cause the output to be directed to your terminal (**stdout**). It is probably a good idea to direct it to a file or to a printer so you can refer to it.

```
 8 main(argc, argv)
23   if (argc < 2)
     /* argc == 2 */
30   opterr = FALSE;
     /* FALSE == 0 */
     /* opterr == 0 */
31   while ((ch = getopt(argc,argv,"opr")) != EOF)
     /* argc == 2 */
     /* argv == 15729316 */
     /* ch == 111 or 'o' or "t" */
32   {
33         switch(ch)
           /* ch == 111 or 'o' or "t" */
35         case 'o':
36               oflag = TRUE;
                 /* TRUE == 1 or "h" */
                 /* oflag == 1 or "h" */
37               break;
48   }
31   while ((ch = getopt(argc,argv,"opr")) != EOF)
     /* argc == 2 */
     /* argv == 15729316 */
     /* ch == 112 or 'p' */
32   {
33         switch(ch)
           /* ch == 112 or 'p' */
38         case 'p':
39               pflag = TRUE;
                 /* TRUE == 1 or "h" */
                 /* pflag == 1 or "h" */
40               break;
48   }
```

Figure 2-20: **ctrace** Output (sheet 1 of 3)

```
31   while ((ch = getopt(argc,argv,"opr")) != EOF)
     /* argc == 2 */
     /* argv == 15729316 */
     /* ch == 114 or 'r' */
32   {
33        switch(ch)
          /* ch == 114 or 'r' */
41        case 'r':
42             rflag = TRUE;
               /* TRUE == 1 or "h" */
               /* rflag == 1 or "h" */
43             break;
48   }
31   while ((ch = getopt(argc,argv,"opr")) != EOF)
     /* argc == 2 */
     /* argv == 15729316 */
     /* ch == -1 */
49   if ((fin = fopen("info","r")) == NULL)
     /* fin == 140200 */
54   if (fscanf(fin, "%s%f%f%f%f%f%f",first.pname,&first.ppx,
     &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
     /* fin == 140200 */
     /* first.pname == 15729528 */
61   printf("Property: %s0,first.pname);
     /* first.pname == 15729528 or "Linden_Place" */ Property: Linden_Place

63   if(oflag)
     /* oflag == 1 or "h" */
64        printf("     Opportunity Cost: $%#5.2f0,oppty(&first));
 5 oppty(ps)
 8   return(ps->i/12 * ps->t * ps->dp);
     /* ps->i == 1069044203 */
     /* ps->t == 1076494336 */
     /* ps->dp == 1088765312 */      Opportunity Cost: $4476.87
```

Figure 2-20: **ctrace** Output (sheet 2 of 3)

```
66  if(pflag)
    /* pflag == 1 or "h" */
67         printf("      Anticipated Profit(loss): $%#7.2f0,pft(&first));
 5 pft(ps)
 8  return(ps->spx - ps->ppx + ps->c);
    /* ps->spx == 1091649040 */
    /* ps->ppx == 1091178464 */
    /* ps->c == 1087409536 */  Anticipated Profit(loss): $85950.00

69  if(rflag)
    /* rflag == 1 or "h" */
70         printf("      Return on Funds Employed: %#3.2f%%0,rfe(&first));
 6 rfe(ps)
 9  return(100 * (ps->spx - ps->c) / ps->spx);
    /* ps->spx == 1091649040 */
    /* ps->c == 1087409536 */  Return on Funds Employed: 94.00%

   /* return */
```

Figure 2-20: **ctrace** Output (sheet 3 of 3)

Using a program that runs successfully is not the optimal way to demonstrate **ctrace**. It would be more helpful to have an error in the operation that could be detected by **ctrace**. It would seem that this utility might be most useful in cases where the program runs to completion, but the output is not as expected.

cxref

cxref analyzes a group of C source code files and builds a cross-reference table of the automatic, static, and global symbols in each file.

The command

cxref -c -o cx.op restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-21 in a file named, in this case, **cx.op**. The **-c** option causes the reports for the four **.c** files to be combined in one cross-reference file.

```
restate.c:

oppty.c:

pft.c:

rfe.c:

  SYMBOL            FILE                     FUNCTION    LINE

 BUFSIZ       /usr/include/stdio.h            --         *9
 EOF          /usr/include/stdio.h            --         49 *50
              restate.c                       --         31
 FALSE        restate.c                       --         *6  15  16  17  30
 FILE         /usr/include/stdio.h            --         *29  73  74
              restate.c                       main       12
 L_ctermid    /usr/include/stdio.h            --         *80
 L_cuserid    /usr/include/stdio.h            --         *81
 L_tmpnam     /usr/include/stdio.h            --         *83
 NULL         /usr/include/stdio.h            --         46 *47
              restate.c                       --         49
 P_tmpdir     /usr/include/stdio.h            --         *82
 TRUE         restate.c                       --         *5  36  39  42
 _IOEOF       /usr/include/stdio.h            --         *41
 _IOERR       /usr/include/stdio.h            --         *42
 _IOFBF       /usr/include/stdio.h            --         *36
 _IOLBF       /usr/include/stdio.h            --         *43
 _IOMYBUF     /usr/include/stdio.h            --         *40
 _IONBF       /usr/include/stdio.h            --         *39
 _IOREAD      /usr/include/stdio.h            --         *37
 _IORW        /usr/include/stdio.h            --         *44
 _IOWRT       /usr/include/stdio.h            --         *38
 _NFILE       /usr/include/stdio.h            --         2 *3  73
 _SBFSIZ      /usr/include/stdio.h            --         *16
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 1 of 5)

```
SYMBOL          FILE                        FUNCTION    LINE
_base           /usr/include/stdio.h        --          *26
_bufend()
                /usr/include/stdio.h        --          *57
_bufendtab      /usr/include/stdio.h        --          *78
_bufsiz()
                /usr/include/stdio.h        --          *58
_cnt            /usr/include/stdio.h        --          *20
_file           /usr/include/stdio.h        --          *28
_flag           /usr/include/stdio.h        --          *27
_iob            /usr/include/stdio.h        --          *73
                restate.c                   main        25 26 45 51
_ptr            /usr/include/stdio.h        --          *21
argc            restate.c                   --          8
                restate.c                   main        *9  23  31
argv            restate.c                   --          8
                restate.c                   main        *10 25 26 31 45 51
c               ./recdef.h                  --          *6
                pft.c                       pft         8
                restate.c                   main        55
                rfe.c                       rfe         9
ch              restate.c                   main        *18  31  33
clearerr()
                /usr/include/stdio.h        --          *67
ctermid()
                /usr/include/stdio.h        --          *77
cuserid()
                /usr/include/stdio.h        --          *77
dp              ./recdef.h                              --*4
                oppty.c                     oppty       8
                restate.c                   main        55
exit()
                restate.c                   main        *13  27  46  52  58
fdopen()
                /usr/include/stdio.h        --          *74
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 2 of 5)

```
  SYMBOL           FILE                 FUNCTION    LINE
feof()
               /usr/include/stdio.h      --         *68
ferror()
               /usr/include/stdio.h      --         *69
fgets()
               /usr/include/stdio.h      --         *77
fileno()
               /usr/include/stdio.h      --         *70
fin            restate.c                 main       *12  49  54
first          restate.c                 main       *19 54 55 61 64 67
fopen()
               /usr/include/stdio.h      --         *74
               restate.c                 main       12  49
fprintf        restate.c                 main       25 26 45 51 57
freopen()
               /usr/include/stdio.h      --         *74
fscanf         restate.c                 main       54
ftell()
               /usr/include/stdio.h      --         *75
getc()
               /usr/include/stdio.h      --         *61
getchar()
               /usr/include/stdio.h      --         *65
getopt()
               restate.c                 main       *14  31
gets()
               /usr/include/stdio.h      --         *77
i              ./recdef.h                --         *5
               oppty.c                   oppty      8
               restate.c                 main       55
lint           /usr/include/stdio.h      --         60
main()
               restate.c                 --         *8
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 3 of 5)

```
 SYMBOL              FILE                  FUNCTION   LINE
oflag              restate.c               main       *15  36  63
oppty()
                   oppty.c                 --         *5
                   restate.c               main       *21  64
opterr             restate.c               main       *20  30
p                  /usr/include/stdio.h    --         *57 *58 *61 62
*62  63  64  67 *67  68 *68  69 *69  70 *70
pdp11              /usr/include/stdio.h    --         11
pflag              restate.c               main       *16  39  66
pft()
                   pft.c                   --         *5
                   restate.c               main       *21  67
pname              ./recdef.h              --         *2
                   restate.c               main       54  61
popen()
                   /usr/include/stdio.h    --         *74
ppx                ./recdef.h              --         *3
                   pft.c                   pft        8
                   restate.c               main       54
printf             restate.c               main       61  64  67  70
ps                 oppty.c                 --         5
                   oppty.c                 oppty      *6  8
                   pft.c                   --         5
                   pft.c                   pft        *6  8
                   rfe.c                   --         6
                   rfe.c                   rfe        *7  9
putc()
                   /usr/include/stdio.h    --         *62
putchar()
                   /usr/include/stdio.h    --         *66
rec                ./recdef.h              --         *1
                   oppty.c                 oppty      6
                   pft.c                   pft        6
                   restate.c               main       19
                   rfe.c                   rfe        7
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 4 of 5)

```
 SYMBOL          FILE                  FUNCTION    LINE
rewind()
                /usr/include/stdio.h   --          *76
rfe()
                restate.c              main        *21  70
                rfe.c                  --          *6
rflag           restate.c              main        *17  42  69
setbuf()
                /usr/include/stdio.h   --          *76
spx             ./recdef.h             --          *8
                pft.c                  pft         8
                restate.c              main        55
                rfe.c                  rfe         9
stderr          /usr/include/stdio.h   --          *55
                restate.c              --          25  26  45  51  57
stdin           /usr/include/stdio.h   --          *53
stdout          /usr/include/stdio.h   --          *54
t               ./recdef.h             --          *7
                oppty.c                oppty       8
                restate.c              main        55
tempnam()
                /usr/include/stdio.h   --          *77
tmpfile()
                /usr/include/stdio.h   --          *74
tmpnam()
                /usr/include/stdio.h   --          *77
u370            /usr/include/stdio.h   --          5
u3b             /usr/include/stdio.h   --          8  19
u3b5            /usr/include/stdio.h   --          8  19
vax             /usr/include/stdio.h   --          8  19
x               /usr/include/stdio.h   --          *62  63  64  66 *66
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 5 of 5)

lint

lint looks for features in a C program that are apt to cause execution errors, that are wasteful of resources, or that create problems of portability.

The command

lint restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-22.

```
restate.c:

restate.c
==============
(71)  warning: main() returns random value to invocation environment
oppty.c:
pft.c:
rfe.c:


==============
function returns value which is always ignored
    printf
```

Figure 2-22: **lint** Output

lint has options that will produce additional information. Check the *IRIS-4D User's Reference Manual.* The error messages give you the line numbers of some items you may want to review.

prof

prof produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. The program must be compiled with the **-p** option. When a program that was compiled with that option is run, a file called **mon.out** is produced. **mon.out** and **a.out** (or whatever name identifies your executable file) are input to the **prof** command.

The sequence of steps needed to produce a profile report for our sample program is as follows:

Step 1: Compile the programs with the **-p** option:

```
cc -p restate.c oppty.c pft.c rfe.c
```

Step 2: Run the program to produce a file **mon.out**.

```
a.out -opr
```

Step 3: Execute the **prof** command:

```
prof a.out
```

The example of the output of this last step is shown in Figure 2-23. The figures may vary from one run to another. You will also notice that programs of very small size, like that used in the example, produce statistics that are not overly helpful.

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
50.0	0.03	0.03	3	8.	fcvt
20.0	0.01	0.04	6	2.	atof
20.0	0.01	0.05	5	2.	write
10.0	0.00	0.05	1	5.	fwrite
0.0	0.00	0.05	1	0.	monitor
0.0	0.00	0.05	1	0.	creat
0.0	0.00	0.05	4	0.	printf
0.0	0.00	0.05	2	0.	profil
0.0	0.00	0.05	1	0.	fscanf
0.0	0.00	0.05	1	0.	_doscan
0.0	0.00	0.05	1	0.	oppty
0.0	0.00	0.05	1	0.	_filbuf
0.0	0.00	0.05	3	0.	strchr
0.0	0.00	0.05	1	0.	strcmp
0.0	0.00	0.05	1	0.	ldexp
0.0	0.00	0.05	1	0.	getenv
0.0	0.00	0.05	1	0.	fopen
0.0	0.00	0.05	1	0.	_findiop
0.0	0.00	0.05	1	0.	open
0.0	0.00	0.05	1	0.	main
0.0	0.00	0.05	1	0.	read
0.0	0.00	0.05	1	0.	strcpy
0.0	0.00	0.05	14	0	ungetc
0.0	0.00	0.05	4	0.	_doprnt
0.0	0.00	0.05	1	0.	pft
0.0	0.00	0.05	1	0.	rfe
0.0	0.00	0.05	4	0.	_xflsbuf
0.0	0.00	0.05	1	0.	_wrtchk
0.0	0.00	0.05	2	0.	_findbuf
0.0	0.00	0.05	2	0.	isatty
0.0	0.00	0.05	2	0.	ioctl
0.0	0.00	0.05	1	0.	malloc
0.0	0.00	0.05	1	0.	memchr
0.0	0.00	0.05	1	0.	memcpy
0.0	0.00	0.05	2	0.	sbrk
0.0	0.00	0.05	4	0.	getopt

Figure 2-23: **prof** Output

size

size produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Here are the results of one invocation of the **size** command with our object file as an argument.

```
11832 + 3872 + 2240 = 17944
```

Don't confuse this number with the number of characters in the object file that appears when you do an **ls -l** command. That figure includes the symbol table and other header information that is not used at run time.

strip

strip removes the symbol and line number information from a common object file. When you issue this command the number of characters shown by the **ls -l** command approaches the figure shown by the **size** command, but still includes some header information that is not counted as part of the .text, .data, or .bss section. After the **strip** command has been executed, it is no longer possible to use the file with the **dbx** or **edge** command.

Program-Organizing Utilities

The following three utilities are helpful in keeping your programming work organized effectively.

The make Command

When you have a program that is made up of more than one module of code you begin to run into problems of keeping track of which modules are up to date and which need to be recompiled when changes are made in another module. The **make** command is used to ensure that dependencies between modules are recorded so that changes in one module results in the re-compilation of dependent programs. Even control of a program as simple as the one shown in Figure 2-15 is made easier through the use of **make**.

The **make** utility requires a description file that you create with an editor. The description file (also referred to by its default name: **makefile**) contains the information used by **make** to keep a target file current. The target file is typically an executable program. A description file contains three types of information:

dependency information	tells the **make** utility the relationship between the modules that comprise the target program.
executable commands	generate the target program. **make** uses the dependency information to determine which executable commands should be passed to the shell for execution.
macro definitions	provide a shorthand notation within the description file to make maintenance easier. Macro definitions can be overridden by information from the command line when the **make** command is entered.

The **make** command works by checking the "last changed" time of the modules named in the description file. When **make** finds a component that has been changed more recently than modules that depend on it, the specified commands (usually compilations) are passed to the shell for execution.

The **make** command takes three kinds of arguments: options, macro definitions, and target file names. If no description file name is given as an option on the command line, **make** searches the current directory for a file named **makefile** or **Makefile**. Figure 2-24 shows a **makefile** for our sample program.

```
OBJECTS = restate.o oppty.o pft.o rfe.o
all: restate
restate: $(OBJECTS)
	$(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o restate

$(OBJECTS): ./recdef.h

clean:
	rm -f $(OBJECTS)

clobber:  clean
	rm -f restate
```

Figure 2-24: **make** Description File

The following things are worth noticing in this description file:

- It identifies the target, **restate**, as being dependent on the four object modules. Each of the object modules in turn is defined as being dependent on the header file, **recdef.h**, and by default, on its corresponding source file.
- A macro, OBJECTS, is defined as a convenient shorthand for referring to all of the component modules.

Whenever testing or debugging results in a change to one of the components of **restate**, for example, a command such as the following should be entered:

make CFLAGS=-g restate

This has been a very brief overview of the **make** utility. There is more on **make** in Chapter 3, and a detailed description of **make** can be found in Chapter 10.

The Archive File

The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the **cc** command line). This causes the link editor to search the symbol table of the archive file when attempting to resolve references.

The **ar** command is used to create an archive file, to manipulate its contents and to maintain its symbol table. The structure of the **ar** command is a little different from the normal UNIX system arrangement of command line options. When you enter the **ar** command you include a one-character key from the set **drqtpmx** that defines the type of action you intend. The key may be combined with one or more additional characters from the set **vuaibcls** that modify the way the requested operation is performed. The makeup of the command line is

ar -key [*posname*] *afile* [*name*]...

where *posname* is the name of a member of the archive and may be used with some optional key characters to make sure that the files in your archive are in a particular order. The *afile* argument is the name of your archive file. By convention, the suffix **.a** is used to indicate the named file is an archive file. (**libc.a**, for example, is the archive file that contains many of the object files of the standard C subroutines.) One or more *names* may be furnished. These identify files that are subjected to the action specified in the *key*.

We can make an archive file to contain the modules used in our sample program, **restate**. The command to do this is

ar -rv rste.a restate.o oppty.o pft.o rfe.o

If these are the only **.o** files in the current directory, you can use shell metacharacters as follows:

ar -rv rste.a *.o

Either command will produce this feedback:

```
a - restate.o
a - oppty.o
a - pft.o
a - rfe.o
ar: creating rste.a
```

The **nm** command is used to get a variety of information from the symbol table of common object files. The object files can be, but don't have to be, in an archive file. Figure 2-25 shows the output of this command when executed with the **-f** (for full) option on the archive we just created. The object files were compiled with the **-g** option.

Symbols from rste.a[restate.o]

Name	Value	Class	Type	Size	Line	Section
.0fake			strtag	struct	16	
restate.c		file				
_cnt	0	strmem	int			
_ptr	4	strmem	*Uchar			
_base	8	strmem	*Uchar			
_flag	12	strmem	char			
_file	13	strmem	char			
.eos		endstr		16		
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
main	0	extern	int( )	520		.text
.bf	10	fcn			11	.text
argc	0	argm't	int			
argv	4	argm't	**char			
fin	0	auto	*struct-.0fake	16		
oflag	4	auto	int			
pflag	8	auto	int			
rflag	12	auto	int			
ch	16	auto	int			

Figure 2-25: **nm** Output, with **-f** Option (sheet 1 of 5)

Symbols from rste.a[restate.o]

Name	Value	Class	Type	Size	Line	Section
first	20	auto	struct-rec	52		
.ef	518	fcn			61	.text
FILE		typdef	struct-.0fake	16		
.text	0	static		31	39	.text
.data	520	static			4	.data
.bss	824	static				.bss
_iob	0	extern				
fprintf	0	extern				
exit	0	extern				
opterr	0	extern				
getopt	0	extern				
fopen	0	extern				
fscanf	0	extern				
printf	0	extern				
oppty	0	extern				
pft	0	extern				
rfe	0	extern				

Figure 2-25: **nm** Output, with **-f** Option (sheet 2 of 5)

Symbols from rste.a[oppty.o]

Name	Value	Class	Type	Size	Line	Section
oppty.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
oppty	0	extern	float()	64		.text
.bf	10	fcn			7	.text
ps	0	argm't	*struct-rec	52		
.ef	62	fcn			3	.text
.text	0	static		4	1	.text
.data	64	static				.data
.bss	72	static				.bss

Figure 2-25: **nm** Output, with **-f** Option (sheet 3 of 5)

Symbols from rste.a[pft.o]

Name	Value	Class	Type	Size	Line	Section
pft.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
..eos		endstr		52		
pft	0	extern	float()	60		.text
..bf	10	fcn			7	.text
ps	0	argm't	*struct-rec	52		
..ef	58	fcn			3	.text
..text	0	static		4		.text
..data	60	static				.data
..bss	60	static				.bss

Figure 2-25: **nm** Output, with **-f** Option (sheet 4 of 5)

Symbols from rste.a[rfe.o]

Name	Value	Class	Type	Size	Line	Section
rfe.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
rfe	0	extern	float()	68		.text
.bf	10	fcn			8	.text
ps	0	argm't	*struct-rec	52		
.ef	64	fcn			3	.text
.text	0	static		4	1	.text
.data	68	static				.data
.bss	76	static				.bss

Figure 2-25: **nm** Output, with **-f** Option (sheet 5 of 5)

For **nm** to work on an archive file all of the contents of the archive have to be object modules. If you have stored other things in the archive, you will get the message:

```
nm: rste.a  bad magic
```

when you try to execute the command.

Use of SCCS by Single-User Programmers

The UNIX system Source Code Control System (SCCS) is a set of programs designed to keep track of different versions of programs. When a program has been placed under control of SCCS, only a single copy of any one version of the code can be retrieved for editing at a given time. When program code is changed and the program returned to SCCS, only the changes are recorded. Each version of the

code is identified by its SID, or SCCS **ID**entifying number. By specifying the SID when the code is extracted from the SCCS file, it is possible to return to an earlier version. If an early version is extracted with the intent of editing it and returning it to SCCS, a new branch of the development tree is started. The set of programs that make up SCCS appear as UNIX system commands. The commands are:

admin
get
delta
prs
rmdel
cdc
what
sccsdiff
comb
val

It is most common to think of SCCS as a tool for project control of large programming projects. It is, however, entirely possible for any individual user of the UNIX system to set up a private SCCS system. Chapter 11 is an SCCS user's guide.

Application Programming Objectives

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on a UNIX system computer.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially for your own private use (we have called this single-user programming), to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

The next three sections of the chapter deal with a number of loosely-related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 2, Programming Basics, but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

Advanced Programming	deals with such topics as file and record locking, interprocess communication, and programming terminal screens.
Support Tools	covers the Common Object File Format, link editor directives, **edge**, and **lint**.
Project Control Tools	includes some discussion of **make** and SCCS.

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

Application Environment Characteristics

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a fairly simple application.

When more than one programmer works on a project, there is a need to share such information as:

- the operation of each function
- the number, identity, and type of arguments expected by a function
- if pointers are passed to a function, are the objects being pointed to modified by the called function, and what is the lifetime of the pointed-to object
- the data type returned by a function

In an application, there is an odds-on possibility that the same function can be used in many different programs, by many different programmers. The object code needs to be kept in a library accessible to anyone on the project who needs it.

Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, a desirable objective often is to produce code that will run on many different UNIX system computers. Some of the things that affect portability will be touched on later in this chapter.

Documentation

A single-user program has modest needs for documentation. There should be enough to remind the program's creator how to use it, and what the intent was in portions of the code.

On an application development project there is a significant need for two types of internal documentation:

- comments throughout the source code that enable successor programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of 5 or more years, and frequently need to be modified during that time. It is not realistic to expect that the same person who wrote the program will always be available to make modifications. Even if that does happen the comments will make the maintenance job a lot easier.
- hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in the same function being written over again.

Language Selection

In this section we talk about some of the considerations that influence the selection of programming languages, and describe two of the special-purpose languages that are part of the UNIX system environment.

Influences

In single-user programming the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when making the same decision for an application development project.

- Is there an existing standard within the organization that should be observed?

  A firm may decide to emphasize one language because a good supply of programmers is available who are familiar with it.

- Does one language have better facilities for handling the particular algorithm?

  One would like to see all language selection based on such objective criteria, but it is often necessary to balance this against the skills of the organization.

- Is there an inherent compatibility between the language and the UNIX operating system?

  This is sometimes the impetus behind selecting C for programs destined for a UNIX system machine.

- Are there existing tools that can be used?

  If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application needs.

- Does the application integrate other software into the whole package?

  If, for example, a package is to be built around an existing data base management system, there may be constraints on the variety of languages the database management system can accommodate.

Special-Purpose Languages

The UNIX system contains a number of tools that can be included in the category of special-purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

The awk Utility

The **awk** utility scans an ASCII input file record by record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the program. The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs consist of only a few. The set of statements may include definitions of the pattern that separates one record from another (a newline character, for example), and what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read, and other actions to be performed after the final record has been read. All statements in between are evaluated in order for each record in the input file. To paraphrase the action of a simple **awk** program, it would go something like this:

> Look through the input file.
> Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

> First do some initialization.
> Then, look through the input file.
> Every time you see this specific pattern, do this action.
> Every time you see this other pattern, do another action.
> After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written very quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

Using awk

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order and pass the resulting rearranged data to a function that adds records to your database. There is an example of a use of **awk** in the sample application at the end of this chapter.

The manual page for **awk** is in Section (1) of the *IRIS-4D User's Reference Manual.* Chapter 4 in Part 2 of this guide contains a description of the **awk** syntax and a number of examples showing ways in which **awk** may be used.

The lex and yacc Utilities

lex and **yacc** are often mentioned in the same breath because they perform complementary parts of what can be viewed as a single task: making sense out of input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be quite similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The UNIX system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing, and that's where **lex** and **yacc** come in. In both utilities, the specifications cause the generation of C language subroutines that deal with streams of characters. **lex** generates subroutines that do lexical analysis, or identify the words of vocabulary of a language as distinguished from its grammar or structure. **yacc** generates subroutines that do parsing by describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

Of course, the important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link edited with others to produce programs that then perform whatever process follows from the recognition of the input.

Using lex

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a token. The rules are stated in a format that closely resembles the one used by the UNIX system text editor for regular expressions. For example,

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of that rule might have this notation:

```
[ \t]+ ;
```

which, in effect, says to ignore white space. It carries this meaning because no action is specified when a string of one or more blanks or tabs is recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+ {
        i = atoi(yytext);
        return(NBR);
        }
```

This rule depends on several things:

NBR must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is **#include**'d in the declaration section.)

i is declared as an **extern int** in the declaration section.

It is a characteristic of **lex** that things it finds are made available in a character string called **yytext.**

Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is **lex** saying, "Hey, I found the kind of token we call NBR, and its value is now in **i.**"

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)

2. **lex.yy.c** is transformed by the C compiler (with a **–c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. **lex.yy.o** is link edited with other subroutines. Presumably one of those subroutines will call **yylex()** with a statement such as:

```
while((token = yylex()) != 0)
```

and other subroutines (or even **main**) will deal with what comes back.

The manual page for **lex** is in Section (1) of the *IRIS-4D Programmer's Reference Manual*. A tutorial on **lex** is contained in Chapter 5 in Part 2 of this guide.

Using yacc

yacc subroutines are produced by pretty much the same series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. **y.tab.c** is compiled by the C compiler producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.
3. The object file **y.tab.o** may be link edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by UNIX system editors. **yacc** rules are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined further down the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. An example might make that easier to understand:

```
%token   NUMBER
%%
expr     : numb                         { $$ = $1; }
         | expr '+' expr                { $$ = $1 + $3; }
         | expr '-' expr                { $$ = $1 - $3; }
         | expr '*' expr                { $$ = $1 * $3; }
         | expr '/' expr                { $$ = $1 / $3; }
         | '(' expr ')'                 { $$ = $2; }
         ;
numb     : NUMBER              { $$ = $1; }
         ;
```

This fragment of a **yacc** specification shows

- NUMBER identified as a token in the declaration section
- the start of the rules section indicated by the pair of percent signs
- a number of alternate definitions for *expr* separated by the | sign and terminated by the semicolon
- actions to be taken when a rule is matched
- within actions, numbered variables used to represent components of the rule:

  $$ means the value to be returned as the value of the whole rule

  $*n* means the value associated with the *n*th component of the rule, counting from the left
- *numb* defined as meaning the token NUMBER. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link edited with other subroutines that handle processing that takes place after the parsing—or even ahead of it.

The manual page for **yacc** is in Section (1) of the *IRIS-4D Programmer's Reference Manual*. A detailed description of **yacc** may be found in Chapter 6 of this guide.

Advanced Programming Tools

In Chapter 2 we described the use of such basic elements of programming in the UNIX system environment as the standard I/O library, header files, system calls and subroutines. In this section we introduce tools that are more apt to be used by members of an application development team than by a single-user programmer. The section contains material on the following topics:

- memory management
- file and record locking
- interprocess communication
- programming terminal screens

Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that a number of records have been extracted from a database and need to be held for some further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it is likely to be a great deal more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished; but such an occurrence is not pertinent to this discussion.) There are two C language subroutines available for acquiring blocks of memory and they are both called **malloc**. One of them is **malloc**(3C), the other is **malloc**(3X). Each has several related commands that do specialized tasks in the same area. They are:

- **free**—to inform the system that space is being relinquished
- **realloc**—to change the size and possibly move the block
- **calloc**—to allocate space for an array and initialize it to zeros

In addition, **malloc**(3X) has a function, **mallopt**, that provides for control over the space allocation algorithm, and a structure, **mallinfo**, from which the program can get information about the usage of the allocated space.

malloc(3X) runs faster than the other version. It is loaded by specifying

-lmalloc

on the **cc**(1) or **ld**(1) command line to direct the link editor to the proper library. When you use **malloc**(3X) your program should contain the statement

```
#include <malloc.h>
```

where the values for **mallopt** options are defined.

See the *IRIS-4D Programmer's Reference Manual* for the formal definitions of the two **mallocs**.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember, in the UNIX system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it, that is, changing any of the data. If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term shared lock is sometimes applied to read locks.

Another distinction needs to be made between mandatory and advisory locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write, or create files. This is done through a permission flag established by the file's owner (or the super-user). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed. Thus, mandatory may sound like a simpler and better deal, but it isn't so. The mandatory locking capability is included in the system to comply with an agreement with */usr/group*, an organization that represents the interests of UNIX system users. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term atomic is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

How File and Record Locking Works

The system call for file and record locking is **fcntl**(2). Programs should include the line

```
#include <fcntl.h>
```

to bring in the header file shown in Figure 3-1.

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define     O_RDONLY    0
#define     O_WRONLY    1
#define     O_RDWR      2
#define     O_NDELAY    04        /* Non-blocking I/O */
#define     O_APPEND    010       /* append (writes guaranteed at the end) */
#define     O_SYNC                020/* synchronous write option */
/* Flag values accessible only to open(2) */
#define     O_CREAT     00400     /* open with file create (uses 3rd open arg)*/
#define     O_TRUNC     01000     /* open with truncation */
#define     O_EXCL      02000     /* exclusive open */
/* fcntl(2) requests */
#define     F_DUPFD     0         /* Duplicate fildes */
#define     F_GETFD     1         /* Get fildes flags */
#define     F_SETFD     2         /* Set fildes flags */
#define     F_GETFL     3         /* Get file flags */
#define     F_SETFL     4         /* Set file flags */
#define     F_GETLK     5         /* Get file lock */
#define     F_SETLK     6         /* Set file lock */
#define     F_SETLKW    7         /* Set file lock and wait */
#define     F_CHKFL     8         /* Check legality of file flag changes */
/*file segment locking set data type - info passed to system by user*/
struct flock {
            short       l_type;
            short       l_whence;
            long        l_start;
            long        l_len;    /* len = 0 means until end of file */
            short       l_sysid;
            short       l_pid;
};
/* file segment locking types */
            /* Read lock */
#define     F_RDLCK     01
            /* Write lock */
#define     F_WRLCK     02
            /* Remove lock(s) */
#define     F_UNLCK     03
```

Figure 3-1: The **fcntl.h** Header File

The format of the **fcntl**(2) system call is

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

fildes is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h** includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

lockf

A subroutine, **lockf**(3), can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

fildes is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek**(2) system call.)

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl**(2), **fcntl**(5), **lockf**(3), and **chmod** (2) in the *IRIS-4D Programmer's Reference Manual*. Chapter 7 in Part 2 of this guide is a detailed discussion of the subject with a number of examples.

Interprocess Communications

In Chapter 2 we described **forking** and **execing** as methods of communicating between processes. Business applications running on a UNIX system computer often need more sophisticated methods. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. To go back to the ticket reservation example again for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, "Yes, sir. Just hang on a minute while I start up the reservations program."

In transaction-driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type the UNIX system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

messages — communication is in the form of data stored in a buffer. The buffer can be either sent or received.

semaphores — communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25.

shared memory — communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

The sets of IPC system calls are:

msgget	**semget**	**shmget**
msgctl	**semctl**	**shmctl**
msgop	**semop**	**shmop**

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET) and removing (IPC_RMID), the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **msgop** has calls that send or receive messages. **semop** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **shmop** has calls that attach or detach shared memory segments.

An example of the use of some IPC features is included in the sample application at the end of this chapter. The system calls are all located in Section (2) of the *IRIS-4D Programmer's Reference Manual*. Don't overlook **intro**(2). It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is contained in Chapter 8 in Part 2 of this guide.

Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the UNIX system. The first of these, **terminfo**, is a database of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** database normally begins at the directory **/usr/lib/terminfo**. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the entry for a Teletype 5425 is located in both the file **/usr/lib/terminfo/5/5425** and the file **/usr/lib/terminfo/t/tty5425**.

Describing the capabilities of a terminal can be a painstaking task. Quite a good selection of terminal entries is included in the **terminfo** database that comes with your computer. However, if you have a type of terminal that is not already described in the database, the best way to proceed is to find a description of one that comes close to having the same capabilities as yours and building on that one. There is a routine (**setupterm**) in **curses**(3X) that can be used to print out descriptions from the database. Once you have worked out the code that describes the capabilities of your terminal, the **tic**(1M) command is used to compile the entry and add it to the database.

curses

After you have made sure that the operating capabilities of your terminal are a part of the **terminfo** database, you can then proceed to use the routines that make up the **curses**(3X) package to create and manage screens for your application.

The **curses** library includes functions to:

- define portions of your terminal screen as windows
- define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal
- read input from a terminal screen into a program
- write output from a program to your terminal screen
- manipulate the information in a window in a virtual screen area and then send it to your physical screen

In the sample application at the end of this chapter, we show how you might use **curses** routines. Chapter 9 in Part 2 of this guide contains a tutorial on the subject. The manual pages for **curses** are in Section (3X), and those for **terminfo** are in Section (4) of the *IRIS-4D Programmer's Reference Manual.*

Programming Support Tools

This section covers UNIX system components that are part of the programming environment, but that have a highly specialized use.

Link Edit Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. The default locations for the standard Common Object File Format sections are described in **a.out**(4) in the *IRIS-4D Programmer's Reference Manual.*

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among a lot of other functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Well, the truth is that in the majority of cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two, possibly related, sets of circumstances.

1. Your application is large and consists of a lot of object files.
2. The hardware your application is to run on is tight for space.

For further information on the Link Edit Command Language see the *IRIS-4D Series Compiler Guide.*

Common Object File Format

A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as setting up archive libraries and using the Symbolic Debugger.

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<syms.h>	symbol table format
<linenum.h>	line number entries
<ldfcn.h>	COFF access routines
<filehdr.h>	file header for a common object file
<a.out.h>	common assembler and link editor output
<scnhdr.h>	section header for a common object file
<reloc.h>	relocation information for a common object file
<storclass.h>	storage classes for common object files

The object file access routines are described below under the heading "The Object File Library."

For further information on the Common Object File Format see the *Assembly Language Programmer's Guide.*

Libraries

A library is a collection of related object files and/or declarations that simplify programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common database can keep the I/O routines in a library that is searched at link edit time.

In Chapter 2 we described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries, the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, see the *Assembly Language Programmer's Guide.* This library consists of several portions. The functions reside in **/lib/libmld.a** and are loaded during the compilation of a C language program by the **-l** command line option:

cc *file* **–lmld**

which causes the link editor to search the object file library. The argument **-lmld** must appear after all files that reference functions in **libmld.a**.

The following header files must be included in the source code.

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

Function	Reference	Brief Description
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.
ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.

Function	Reference	Brief Description
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek**(3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen**(3X)	Open object file for reading.
ldrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread**(3X)	Read section header of an object file being processed.
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.

Function	Reference	Brief Description
ldtbindex	**ldtbindex**(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread**(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek**(3X)	Seeks to the symbol table of the object file being processed.
sgetl	**sputl**(3X)	Access long integer data in a machine independent format.
sputl	**sputl**(3X)	Translate a long integer into a machine independent format.

Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type LDFILE, which is in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure. The fields of the LDFILE structure may be accessed individually through the following macros:

- The TYPE macro returns the magic number of the file, which is used to distinguish between archive files and object files that are not part of an archive.
- The IOPTR macro returns the file pointer, which was opened by **ldopen**(3X) and is used by the input/output functions of the C library.
- The OFFSET macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.

- The HEADER macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *IRIS-4D Programmer's Reference Manual.*

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the **–l** option on a command line, as follows:

cc *file* **–lm**

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of each file that uses the routines.

The functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double-precision.

Function	Reference	Brief Description
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0**, **j1**, **jn** , **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

Function	Reference	Brief Description
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Function	Reference	Brief Description
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot**(3M)	Return the square root of the sum of the squares of two numbers.

Function	Reference	Brief Description
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the logarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

A Basic Lesson on Debugging

A debugger helps you find bugs in your executable files, and points you to the corresponding errors in your source files. Specifically, it lets you:

- stop your program at specified points to check current values
- trace variables as they change throughout your program
- step through functions one line at a time

And, most importantly, you can use a debugger without modifying your source code.

How Does edge Work?

To use edge, you don't need to change your source file, but you do need to add debugging information to your executable file. This is a simple task – just use the edge flag (**-g**) when you compile your source code. This flag tells the compiler to create an expanded symbol table in the object file. The result is an executable file that contains extra information about your program for the debugger.

In addition to the expanded executable file, edge uses the UNIX *core file* as a source of information about your program. This file is a picture of your program when it faulted.

edge combines these two resources to produce usable, source code level information that helps you debug your programs.

About the edge Environment

edge is a window-based, graphical version of dbx, a standard UNIX debugger. The edge environment consists of three windows that display the dbx system prompt, the source code that you are debugging, and the results that you see when you run the program (standard in, standard out, and standard error). You issue commands to edge through both the keyboard and the mouse.

edge offers many advantages:

- You can watch your source code as it executes in one window, and watch the results in another window.
- Using the mouse to select text and menu choices rather than typing commands speeds up your debugging session.
- The graphic interface makes it easier to learn.

edge runs under the Silicon Graphics, Inc. window manager, **mex**. Because of this, you must be running **mex** to use **edge**.

If you have never used **mex**, it would be a good idea to look over chapter 1 of *Getting Started with your IRIS-4D Series Workstation*, or the *Graphics Library User's Guide* chapter on using mex. This will give you a basic understanding of the **mex** interface. For more information on **edge**, see *Learning to Debug with* **edge**.

lint as a Portability Tool

It is a characteristic of the UNIX system that language compilation systems are somewhat permissive. Generally speaking it is a design objective that a compiler should run fast. Most C compilers, therefore, let some things go unflagged as long as the language syntax is observed statement by statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, there may be real difficulties in running it on some other machine.

That's where **lint** comes in. **lint** produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- cases of disagreement between the type of value expected from a called function and what the function actually returns
- disagreement between the types and number of arguments expected by functions and what the function receives
- inconsistencies that might prove to be bugs
- things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint**.

Code such as this:

```
int i = lseek(fdes, offset, whence)
```

would get by most compilers. However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, it would produce incorrect results, because **i** would contain only the last 16 bits of the value returned.

Since it is reasonable to expect that an application written for a UNIX system machine will be able to run on a variety of computers, it is important that the use of **lint** be a regular part of the application development.

Chapter 12 in Part 2 of this guide contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *IRIS-4D Programmer's Reference Manual.*

Project Control Tools

Volumes have been written on the subject of project control. It is an item of top priority for the managers of any application development team. Two UNIX system tools that can play a role in this area are described in this section.

make

make is extremely useful in an application development project for keeping track of what object files need to be recompiled as changes are made to source code files. One of the characteristics of programs in a UNIX system environment is that they are made up of many small pieces, each in its own object file, that are link edited together to form the executable file. Quite a few of the UNIX system tools are devoted to supporting that style of program architecture. For example, archive libraries, and the fact that the **cc** command accepts **.o** files as well as **.c** files, and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**, are all important elements of modular architecture. The two main advantages of this type of programming are that

- A file that performs one function can be re-used in any program that needs it.
- When one function is changed, the whole program does not have to be recompiled.

On the flip side, however, a consequence of the proliferation of object files is an increased difficulty in keeping track of what does need to be recompiled, and what doesn't. **make** is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Once having done that, you conclude a session in which possibly a number of your source code files have been changed by running the **make** command. **make** takes care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

make has the ability to work with files in archive libraries or under control of the Source Code Control System (SCCS).

The **make**(1) manual page is contained in the *IRIS-4D Programmer's Reference Manual.* Chapter 10 in Part 2 of this guide gives a complete description of how to use **make.**

SCCS

SCCS is an acronym for Source Code Control System. It consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can:

- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for tasks such as editing and compiling. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program, **restate**, that was used in several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```
                    /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
                    return(ps->i/12 * ps->t * ps->dp);
}
```

If you decide to add a message to this funtion, you might change the file like this:

```
                                        /* Opportunity Cost -- oppty.c */
#include "recdef.h"
#include <stdio.h>

float
oppty(ps)
struct rec *ps;
{
                                        (void) fprintf(stderr, "Opportunity calling\n");
                                        return(ps->i/12 * ps->t * ps->dp);
}
```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change, and the login id of the person making the change. Chapter 11 in Part 2 of this guide is an SCCS user's guide. SCCS commands are in Section (1) of the *IRIS-4D Programmer's Reference Manual.*

liber, A Library System

To illustrate the use of UNIX system programming tools in the devĕlopment of an application, we are going to pretend we are engaged in the development of a computer system for a library. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. It is the way it is only in order to provide some passably realistic examples of UNIX system programming tools in use.

liber is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the database. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When the system is running the index resides in memory. Semaphores are used to control access to the index. In the pages that follow fragments of some of the system's programs are shown to illustrate the way they work together. The startup program performs the system initialization; opening the semaphores and shared memory; reading the index into the shared memory; and kicking off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

```
/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
        (void) fprintf(stderr, "shmat failed: %d\n",errno);
        exit(1);
}
```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and the book cannot be checked out at two different checkout stations at the same time.

The program fragments do not provide any details on the structure of the index or the book records in the database.

```
                          /* liber.h - header file for the
                           *          library system.
                           */
typedef ... INDEX;        /* data structure for book file index */
typedef struct {          /* type of records in book file */
      char title[30];
      char author[30];
      .
      .
      .
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;
```

```
/*   startup program */

/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write access to index.
 * 3. Stash id's for shared memory segment and semaphores in a file
 *    where they can be accessed by the programs.
 * 4. Start programs:  add-books, card-catalog, and checkout running
 *    on the various terminals throughout the library.
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <sys/sem.h>
#include   "liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;

main()
{
      .
      .
      .
      if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT | 0666)) == -1)
      {
            (void) fprintf(stderr,"startup: shmget failed: errno=%d\n",errno);
            exit(1);
      }
      if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
      {
            (void) fprintf(stderr,"startup: semget failed: errno=%d\n",errno);
            exit(1);
      }
```

```
        if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
        {
              (void) fprintf(stderr,"startup: semget failed: errno=%d\n",errno);
              exit(1);
        }
        (void) fprintf(ipc_file, "%d\n%d\n%d\n", shmid, wrtsem, rdsem);

        /* Start the add-books program running on the terminal in the
         * basement.  Start the checkout and card-catalog programs
         * running on the various other terminals throughout the library.
         */
        .
        .
        .
}

/*    card-catalog program*/

/* 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. Print book record on screen or indicate book was not found.
 * 5. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include    <fcntl.h>
#include    "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];

main() {
        .
        .
        .
```

```
        while (1)
        {
                /* Read author/title/subject information from screen.  */
                /* Wait for write semaphore to reach 0 (index is not written). */
                sop[0].sem_op = 1;
                if (semop(wrtsem, sop, 1) == -1)
                {
                        (void) fprintf(stderr, "semop failed: %d\n", errno);
                        exit(1);
                }
                /* Increment read semaphore so potential writer will wait
                 * for us to finish reading the index.
                 */
                sop[0].sem_op = 0;
                if (semop(rdsem, sop, 1) == -1)
                {
                        (void) fprintf(stderr, "semop failed: %d\n", errno);
                        exit(1);
                }

                /* Use index to find file pointer(s) for book(s) */

                /* Decrement read semaphore */
                sop[0].sem_op = -1;
                if (semop(rdsem, sop, 1) == -1)
                {
                        (void) fprintf(stderr, "semop failed: %d\n", errno);
                        exit(1);
                }

                /* Now we use the file pointers found in the index to
                 * read the book file.  Then we print the information
                 * on the book(s) to the screen.
                 */
        } /* while */
}
/*    checkout program*/

/* 1. Read screen for Dewey Decimal number of book to be checked out.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise lock
 *    book record and read.
 * 5. If book already checked out print message on screen, otherwise
```

```
 *    mark record "checked out" and write back to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include   <fcntl.h>
#include   "liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;

main()
{
     .
     .
     .
     while (1)
     {
          /*
           * Read Dewey Decimal number from screen.
           */
```

```
/*
 * Wait for write semaphore to reach 0 (index not being written).
 */
sop[0].sem_flg = 0;
sop[0].sem_op = 0;
if (semop(wrtsem, sop, 1) == -1)
{
          (void) fprintf(stderr, "semop failed: %d\n", errno);
          exit(1);
}
/*
 * Increment read semaphore so potential writer will wait
 * for us to finish reading the index.
 */
sop[0].sem_op = 1;
if (semop(rdsem, sop, 1) == -1)
{
          (void) fprintf(stderr, "semop failed: %d\n", errno);
          exit(1);
}

/*
 * Now we can use the index to find the book's record position.
 * Assign this value to "bookpos".
 */

/* Decrement read semaphore */
sop[0].sem_op = -1;
if (semop(rdsem, sop, 1) == -1)
{
          (void) fprintf(stderr, "semop failed: %d\n", errno);
          exit(1);
}

/* Lock the book's record in book file, read the record. */
flk.l_type = F_WRLCK;
flk.l_whence = 0;
flk.l_start = bookpos;
flk.l_len = sizeof(BOOK);
if (fcntl(book_file, F_SETLKW, &flk) == -1)
```

```
            {
                        (void) fprintf(stderr, "trouble locking:%d\n", errno);
                        exit(1);
            }
            if (lseek(book_file, bookpos, 0) == -1)
            {
                        Error processing for lseek;
            }
            if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
            {
                        Error processing for read;
            }

            /*
             * If the book is checked out inform the client, otherwise
             * mark the book's record as checked out and write it
             * back into the book file.
             */

            /* Unlock the book's record in book file. */
            flk.l_type = F_UNLCK;
            if (fcntl(book_file, F_SETLK, &flk) == -1)
            {
                        (void) fprintf(stderr, "trouble unlocking:%d\n", errno);
                        exit(1);
            }
      } /* while */
}

/*    add-books program*/

/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
 * 3. Use semaphore "wrtsem" to block new readers.
 * 4. Wait for semaphore "rdsem" to reach 0.
 * 5. Insert book into index.
 * 6. Decrement wrtsem.
 * 7. Go to 1.
 */
```

```
#include <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>
#include   "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
      .
      .
      .
      for (;;)
      {

            /*
             * Read information on new book from screen.
             */

            addscr(&bookbuf);

            /* write new record at the end of the bookfile.
             * Code not shown, but
             * addscr() returns a 1 if title information has
             * been entered, 0 if not.
             */

            /*
             * Increment write semaphore, blocking new readers from
             * accessing the index.
             */
            sop[0].sem_flg = 0;
            sop[0].sem_op = 1;
            if (semop(wrtsem, sop, 1) == -1)
            {
                        (void) fprintf(stderr, "semop failed: %d\n", errno);
                        exit(1);
            }
```

```
        /*
         * Wait for read semaphore to reach 0 (all readers to finish
         * using the index).
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Now that we have exclusive access to the index we
         * insert our new book with its file pointer.
         */

        /* Decrement write semaphore, permitting readers to read index. */
        sop[0].sem_op = -1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
    } /* for */
    .
    .
    .
}
```

The example following, **addscr()**, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
                        /*  addscr is called from add-books.
                         *  The user is prompted for title
                         *  information.
                         */
#include <curses.h>

WINDOW *cmdwin;

addscr(bb)
struct BOOK *bb;
{
    int c;

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(6, 40, 3, 20);
    mvprintw(0, 0, "This screen is for adding titles to the database");
    mvprintw(1, 0, "Enter  a  to add;  q  to quit: ");
    refresh();
    for (;;)
    {
        refresh();
        c = getch();
        switch (c) {
          case 'a':
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter title: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->title);
                noecho();
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter author: ");
                wmove(cmdwin, 2, 1);
```

```
                        echo();
                        wrefresh(cmdwin);
                        wgetstr(cmdwin, bb->author);
                        noecho();
                        werase(cmdwin);
                        wrefresh(cmdwin);
                        endwin();
                        return(1);
                case 'q':
                        erase();
                        endwin();
                        return(0);
                }
        }
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
      $(CC) $(CFLAGS) -o startup startup.c

add-books: add-books.o addscr.o
      $(CC) $(CFLAGS) -o add-books add-books.o addscr.o

add-books.o: liber.h

checkout: liber.h checkout.c
      $(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
      $(CC) $(CFLAGS) -o card-catalog card-catalog.c
```

Introduction

NOTE

This chapter describes the new version of **awk**, called **nawk**, released in UNIX System V Release 3.1 and described in **nawk**(1). An earlier version is described in **awk**(1). The new version will become the default in the next major UNIX system release.

Suppose you want to tabulate some survey results stored in a file, print various reports summarizing these results, generate form letters, reformat a data file for one application package to use with another package, or count the occurrences of a string in a file. **nawk** is a programming language that makes it easy to handle these and many other tasks of information retrieval and data processing. The name **nawk** is an acronym; **n** means "new," and **awk** is constructed from the initials of its developers. **nawk** denotes the language and also the UNIX system command you use to run an **nawk** program.

nawk is an easy language to learn. It automatically does quite a few things that you have to program for yourself in other languages. As a result, many useful **nawk** programs are only one or two lines long. Because **nawk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **nawk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **nawk** and is intended to make it easy for you to start writing and running your own **nawk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **nawk** user, there's a summary of the language at the end of the chapter.

You should be familiar with the UNIX system and shell programming to use this chapter. Although you don't need other programming experience, some knowledge of the C programming language is beneficial, because many constructs found in **nawk** are also found in C.

Basic nawk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Program Structure

The basic operation of **nawk(1)** is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **nawk** program is a sequence of pattern-action statements, as Figure 4-1 shows.

Structure:

pattern { *action* }
pattern { *action* }
...

Example:

```
$1 == "address"  { print $2, $3 }
```

Figure 4-1: **nawk** Program Structure and Example

The example in the figure is a typical **nawk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is `address`. In general, **nawk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in:

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in:

```
{ print $1, $2 }
```

the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

There are two ways to run an **nawk** program. First, you can type the command line:

nawk '*pattern-action statements*' *optional list of input files*

to execute the pattern-action statements on the set of named input files. For example, you could say:

nawk '{ print $1, $2 }' file1 file2

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **nawk**(1) reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (–) as one of the input files. For example:

nawk '{ print $3, $4 }' file1 –

says to read input first from **file1** and then from the standard input.

The arrangement above is convenient when the **nawk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **–f** option to fetch it:

nawk –f *program file* *optional list of input files*

For example, the following command line says to fetch and execute **myprogram** on input from the file **file1**:

nawk –f myprogram file1

Fields

nawk normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. **nawk** then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **nawk** programs in this chapter, we use the file **countries**. It contains information about the ten largest countries in the world. Each record contains a country name, its area in thousands of square miles, its population in millions, and the continent on which it is found. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank separates `North` and `South` from `America`.

```
USSR         8650   262   Asia
Canada       3852   24    North America
China        3692   866   Asia
USA          3615   219   North America
Brazil       3286   116   South America
Australia    2968   14    Australia
India        1269   637   Asia
Argentina    1072   26    South America
Sudan        968    19    Africa
Algeria      920    18    Africa
```

Figure 4-2: The Sample Input File **countries**

This file is typical of the kind of data **nawk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so that the first record of **countries** would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs. The first field within a line is called **$1,** the second **$2,** and so forth. The entire record is called **$0.**

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **nawk** program consisting of a single **print** statement:

```
{ print }
```

so the command line:

nawk '{ print }' countries

prints each line of **countries,** copying the file to the standard output. The **print** statement can also be used to print parts of a record; for instance, the program:

```
{ print $1, $3 }
```

prints the first and third fields of each record. Thus:

nawk '{ print $1, $3 }' countries

produces as output the sequence of lines:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which by default is a single blank. Each line printed is terminated by the output record separator, which by default is a newline.

NOTE

In the remainder of this chapter, we only show **nawk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **nawk** command, or by putting it in a file and invoking **nawk** with the **–f** flag, as discussed in "**nawk** Command Usage." In an example, if no input is mentioned, the input is assumed to be the file **countries**.

Formatted Printing

For more carefully formatted output, **nawk** provides a C-like **printf** statement:

printf *format*, $expr_1$, $expr_2$, . . ., $expr_n$

which prints the $expr_i$'s according to the specification in the string *format*. For example, the **nawk** program:

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field ($1) as a string of 10 characters (right justified), then a space, then the third field ($3) as a decimal number in a six-character field, then a newline (\n). With input from the file **countries**, this program prints an aligned table:

```
     USSR    262
   Canada     24
    China    866
      USA    219
   Brazil    116
Australia     14
    India    637
Argentina     26
    Sudan     19
  Algeria     18
```

With **printf,** no output separators or newlines are produced automatically; you must create them yourself by using **\n** in the format specification. "The **printf** Statement" in this chapter contains a full description of **printf.**

Simple Patterns

You can select specific records for printing or other processing by using simple patterns. **nawk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. For example, the operator == tests for equality. To print the lines for which the fourth field equals the string `Asia`, we can use the program consisting of the single pattern:

```
$4 == "Asia"
```

With the file **countries** as input, this program yields:

```
USSR       8650    262    Asia
China      3692    866    Asia
India      1269    637    Asia
```

The complete set of comparisons is >, >=, <, <=, == (equal to) and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with a population greater than 100 million. The program:

```
$3 > 100
```

is all that is needed. (Remember that the third field in the file **countries** is the population in millions.) It prints all lines in which the third field exceeds 100.

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters US in any position; with the file **countries** as input, it prints:

```
USSR        8650     262     Asia
USA         3615     219     North America
```

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, **BEGIN** and **END,** that match before the first record has been read and after the last record has been processed. This program uses **BEGIN** to print a title:

```
BEGIN   { print "Countries of Asia:" }
/Asia/  { print "     ", $1 }
```

The output is:

```
Countries of Asia:
     USSR
     China
     India
```

Simple Actions

We have already seen the simplest action of an **nawk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **nawk**(1) counts the number of records read and the number of fields within the current record; you can use these counts in your **nawk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. So the program:

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while:

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-defined Variables

Besides providing built-in variables like **NF** and **NR, nawk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file **countries:**

```
        { sum = sum + $3 }
END     { print "Total population is", sum, "million"
          print "Average population of", NR, "countries is", sum/NR }
```

NOTE: **nawk** initializes sum to zero before it is used.

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

nawk has built-in functions that handle common arithmetic and string operations for you. For example, there's an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. **nawk** also lets you define your own functions. Functions are described in detail in the section "Actions" in this chapter.

A Handful of Useful One-liners

Although **nawk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. A collection of other short programs follow; you may find them useful and instructive. They are not explained here, but any new constructs are discussed later in this chapter.

Print last field of each input line:

```
{ print $NF }
```

Print 10th input line:

```
NR == 10
```

Print last input line:

```
        { line = $0}
END     { print line }
```

Print input lines that don't have four fields:

```
NF != 4     { print $0, "does not have 4 fields" }
```

Print input lines with more than four fields:

```
NF > 4
```

Print input lines with last field more than 4:

```
$NF > 4
```

Print total number of input lines:

```
END     { print NR }
```

Print total number of fields:

```
        { nf = nf + NF }
END     { print nf }
```

Print total number of input characters:

```
        { nc = nc + length($0) }
END     { print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/  { nlines++ }
END     { print nlines }
```

(The statement `nlines++` has the same effect as `nlines = nlines + 1`.)

Error Messages

If you make an error in your **nawk** program, you generally get an error message. For example, trying to run the program

```
$3 < 200 { print ( $1 }
```

generates the error messages

```
nawk: syntax error at source line 1
 context is
        $3 < 200 { print ( >>> $1 } <<<
nawk: illegal statement at source line 1
         1 extra (
```

Some errors may be detected while your program is running. For example, if you try to divide a number by zero, **nawk** stops processing and reports the input record number (**NR**) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **nawk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once, before the **nawk** command starts to read its first input record. The pattern **END** matches the end of the input, after the last record has been processed.

The following **nawk** program uses **BEGIN** to set the field separator to tab (**\t**) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
               "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
        area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file **countries** as input, this program produces

```
   COUNTRY   AREA   POP   CONTINENT
      USSR   8650   262   Asia
    Canada   3852    24   North America
     China   3692   866   Asia
       USA   3615   219   North America
    Brazil   3286   116   South America
 Australia   2968    14   Australia
     India   1269   637   Asia
 Argentina   1072    26   South America
     Sudan    968    19   Africa
   Algeria    920    18   Africa

     TOTAL  30292  2201
```

Relational Expressions

An **nawk** pattern can be any expression involving comparisons between strings of characters or numbers. **nawk** has six relational operators, and two regular expression matching operators, ~ (tilde) and !~, which are discussed in the next section, for making comparisons. Figure 4-3 shows these operators and their meanings.

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

Figure 4-3: **nawk** Comparison Operators

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **nawk** can tell what is intended. The section "Number or String?" contains more information about this.) Thus, the pattern `$3>100` selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters S through Z, namely,

```
USSR     8650   262    Asia
USA      3615   219    North America
Sudan    968    19     Africa
```

In the absence of any other information, **nawk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia   2968   14   Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

nawk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions, and are like those in **egrep**(1) and **lex**(1). The simplest regular expression is a string of characters enclosed in slashes, such as

```
/Asia/
```

This program prints all input records that contain the substring `Asia`. (If a record contains `Asia` as part of a larger string like `Asian` or `Pan-Asiatic`, it is also printed.) In general, if *re* is a regular expression, then the pattern

/re/

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators ~ (matches) and !~ (does not match). The program

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches `Asia`, while the program

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match `Asia` .

In regular expressions, the symbols

\ ^ $. [ ] * + ? ( ) |

are metacharacters with special meanings like the metacharacters in the UNIX shell. For example, the metacharacters ^ and $ match the beginning and end, respectively, of a string, and the metacharacter . ("dot") matches any single character. Thus,

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, /[ABC]/ matches records containing any one of **A, B,** or **C** anywhere. Ranges of letters or digits can be abbreviated within brackets: /[a-zA-Z]/ matches any single letter.

If the first character after the [is a ^, this complements the class so it matches any character not in the set: / [^**a-zA-Z**] / matches any non-letter. The program

```
$2 !~ /^[0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (^ for beginning of string, `[0-9]+` for one or more digits, and `$` for end of string). Programs of this nature are often used for data validation.

Parentheses () are used for grouping and the symbol | is used for alternatives. To match lines containing any one of the four substrings `apple pie`, `apple tart`, `cherry pie`, or `cherry tart`, use

```
/(apple|cherry) (pie|tart)/
```

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, to print all lines containing `b` followed by a dollar sign, use

```
/b\$/
```

In addition to recognizing metacharacters, **nawk** recognizes the following C programming language escape sequences within regular expressions and strings:

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
ddd	octal value *ddd*
\"	quotation mark
c	any other character *c* literally

For example, to print all lines containing a tab, use the program:

```
/\t/
```

nawk interprets any string or variable on the right side of a ~ or !~ as a regular expression. For example, we could have written the program:

```
$2 !~ /^[0-9]+$/
```

as

```
BEGIN     { digits = "^[0-9]+$" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters like **^ [0-9] + $**. When a literal quoted string like: **"^ [0-9] + $"** is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing `b` followed by a dollar sign. The regular expression for this pattern is **b\$**. If we want to create a string to represent this regular expression, we must add one more backslash: **"b\\$"**. The two regular expressions on each of the following lines are equivalent:

```
x ~ "b\\$"          x ~ /b\$/
x ~ "b\$"           x ~ /b$/
x ~ "b$"            x ~ /b$/
x ~ "\\t"           x ~ /\t/
```

The precise form of regular expressions and the substrings they match is given in Figure 4-4. The unary operators *****, **+**, and **?** have the highest precedence, then concatenation, and then alternation **|** . All operators are left associative. *r* stands for any regular expression.

Expression	Matches
c	any non-metacharacter *c*
\c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
r_1r_2	r_1 then r_2 (concatenation)
$r_1 \| r_2$	r_1 or r_2 (alternation)

Figure 4-4: **nawk** Regular Expressions

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators `||` (or), **&&** (and), and **!** (not). For example, suppose we want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The program

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with `Asia` or `Africa` as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator `|`:

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator **!** has the highest precedence, then **&&**, and finally `||`. The operators **&&** and `||` evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

pat_1, pat_2 { ... }

In this case, the action is performed for each line between an occurrence of pat_1 and the next occurrence of pat_2 (inclusive). As an example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string Canada up through the next occurrence of the string Brazil:

```
Canada          3852        24          North America
China           3692        866         Asia
USA             3615        219         North America
Brazil          3286        116         South America
```

Similarly, since **FNR** is the number of the current record in the current input file (**FILENAME** is the name of the current input file), to print the first five records of each input file with the name of the current input file prepended:

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Figure 4-5 lists the built-in variables that **nawk** maintains. Some of these we have already met; others are used in this and later sections.

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by **match()**	-
RLENGTH	length of string matched by **match()**	-
SUBSEP	subscript separator	"\034"

Figure 4-5: **nawk** Built-in Variables

Arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file **countries**. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression **1000 * $3 / $2** gives the population density in people per square mile. The program

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

applied to the file **countries** prints the name of each country and its population density:

```
      USSR   30.3
    Canada    6.2
     China  234.6
       USA   60.6
    Brazil   35.3
 Australia    4.7
     India  502.0
 Argentina   24.3
     Sudan   19.6
   Algeria   19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +, –, *, /, % (remainder) and ^ (exponentiation; ** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **nawk** recognizes and produces scientific (exponential) notation: `1e6`, `1E6`, `10e5`, and `1000000` are numerically equal.

nawk has assignment statements like those found in the C programming language. The simplest form is the assignment statement

$v = e$

where v is a variable or field name, and e is an expression. For example, to compute the number of Asian countries and their total population, we could write

```
$4 == "Asia"   { pop = pop + $3; n = n + 1 }
END            { print "population of", n,
                       "Asian countries in millions is", pop }
```

Applied to **countries,** this program produces

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **nawk** initializes each variable with the string value "" and the numeric value **0**.

The assignments in the previous program can be written more concisely using the operators += and ++:

```
$4 == "Asia"        { pop += $3; ++n }
```

The operator += is borrowed from the C programming language:

```
pop += $3
```

has the same effect as

```
pop = pop + $3
```

but the += operator is shorter and runs faster. The same is true of the ++ operator, which adds one to a variable.

The abbreviated assignment operators are +=, −=, *=, /=, %=, and ^=. Their meanings are similar:

v op= e

has the same effect as

v = v op e.

The increment operators are ++ and −−. As in C, they may be used as prefix (++x) or postfix (x++) operators. If **x** is 1, then **i=++x** increments **x**, then sets **i** to 2, while **i=x++** sets **i** to 1, then increments **x**. An analogous interpretation applies to prefix and postfix −−.

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3   { maxpop = $3; country = $1 }
END           { print country, maxpop }
```

Note, however, that this program would not be correct if all values of `$3` were negative.

nawk provides the built-in arithmetic functions shown in Figure 4-6.

Function	Value Returned
atan2(y,x)	arctangent of y/x in the range $-\pi$ to π
cos(x)	cosine of x, with x in radians
exp(x)	exponential function of x
int(x)	integer part of x truncated towards 0
log(x)	natural logarithm of x
rand()	random number between 0 and 1
sin(x)	sine of x, with x in radians
sqrt(x)	square root of x
srand(x)	x is new seed for rand()

Figure 4-6: **nawk** Built-in Arithmetic Functions

x and y are arbitrary expressions. The function **rand**() returns a pseudo-random floating point number in the range (0,1), and **srand**(x) can be used to set the seed of the generator. If **srand**() has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in **"abc"** or **"hello, everyone"**. String constants may contain the C programming language escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program

```
{ print NR ":" $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

nawk provides the built-in string functions shown in Figure 4-7. In this table, *r* represents a regular expression (either as a string or as */r/*), *s* and *t* string expressions, and *n* and *p* integers.

Function	Description
gsub(*r*,*s*)	substitute *s* for *r* globally in current record, return number of substitutions
gsub(*r*,*s*,*t*)	substitute *s* for *r* globally in string *t*, return number of substitutions
index(*s*,*t*)	return position of string *t* in *s*, 0 if not present
length(*s*)	return length of *s*
match(*s*,*r*)	return the position in *s* where *r* occurs, 0 if not present
split(*s*,*a*)	split *s* into array *a* on FS, return number of fields
split(*s*,*a*,*r*)	split *s* into array *a* on *r*, return number of fields
sprintf(*fmt*,*expr-list*)	return *expr-list* formatted according to format string *fmt*
sub(*r*,*s*)	substitute *s* for first *r* in current record, return number of substitutions
sub(*r*,*s*,*t*)	substitute *s* for first *r* in *t*, return number of substitutions
substr(*s*,*p*)	return suffix of *s* starting at position *p*
substr(*s*,*p*,*n*)	return substring of *s* of length *n* starting at position *p*

Figure 4-7: **nawk** Built-in String Functions

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(1). The function **gsub**(*r*,*s*,*t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed**, the leftmost match is used, and is made as long as possible.) It returns the number of substitutions made. The function **gsub**(*r*,*s*) is a synonym for **gsub**(*r*,*s*,**$0**). For example, the program

```
{ gsub(/USA/, "United States"); print }
```

transcribes its input, replacing occurrences of `USA` by `United States`. The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index**(*s*,*t*) returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s*. The first character in a string is at position 1. For example,

```
index("banana", "an")
```

returns 2.

The **length** function returns the number of characters in its argument string; thus,

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (**$0** does not include the input record separator.) The program

```
length($1) > max   { max = length($1); name = $1 }
END                { print name }
```

applied to the file **countries** prints the longest country name: `Australia`.

The **match**(*s*,*r*) function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function also sets two built-in variables **RSTART** and **RLENGTH**. **RSTART** is set to the starting position of the match in the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. (If a match does not occur, **RSTART** is 0, and **RLENGTH** is −1.) For example, the following program finds the first occurrence of the letter `i` followed by at most one character followed by the letter `a` in a record:

```
{ if (match($0, /i.?a/))
      print RSTART, RLENGTH, $0 }
```

It produces the following output on the file **countries**:

```
17 2 USSR        8650   262   Asia
26 3 Canada      3852   24    North America
3 3 China        3692   866   Asia
24 3 USA         3615   219   North America
27 3 Brazil      3286   116   South America
8 2 Australia    2968   14    Australia
4 2 India        1269   637   Asia
7 3 Argentina    1072   26    South America
17 3 Sudan       968    19    Africa
6 2 Algeria      920    18    Africa
```

NOTE

match() matches the left-most longest matching string. For example, with the record

```
AsiaaaAsiaaaaan
```

as input, the program

```
{ if (match($0, /a+/))  print RSTART, RLENGTH, $0 }
```

matches the first string of a's and sets RSTART to 4 and RLENGTH to 3.

The function **sprintf**(*format*, $expr_1$, $expr_2$, . . ., $expr_n$) returns (without printing) a string containing $expr_1$, $expr_2$, . . ., $expr_n$ formatted according to the **printf** specifications in the string *format*. "The **printf** Statement" in this chapter contains a complete specification of the format conventions. The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to x the string produced by formatting the values of $1 and $2 as a ten-character string and a decimal number in a field of width at least six; x may be used in any subsequent computation.

The function **substr**(*s,p,n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s,p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in **countries** to their first three characters by invoking the program

```
{ $1 = substr($1, 1, 3); print }
```

on this file to produce

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting $1 in the program forces **nawk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file **countries,**

```
      { s = s substr($1, 1, 3) " " }
END   { print s }
```

prints

```
USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

by building s up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables **$1, $2, ..., $NF**. Field variables share all of the properties of other variables — they may be used in arithmetic or string operations, and they may have values assigned to them. So, for example, you can divide the second field of the file **countries** by 1000 to convert the area from thousands to millions of square miles:

```
{ $2 /= 1000; print }
```

or assign a new string to a field:

```
BEGIN                       { FS = OFS = "\t" }
$4 == "North America"       { $4 = "NA" }
$4 == "South America"       { $4 = "SA" }
                            { print }
```

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the `print` in the fourth line of the program prints the value of **$0** after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, **$(NF–1)** is the second to last field of the current record. The parentheses are needed: the value of **$NF–1** is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, **$(NF+1),** has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file **countries** creates a fifth field giving the population density:

```
BEGIN   { FS = OFS = "\t" }
        { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

`pop` and `$3` must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like

```
print $1 ":" $2
```

`$1` and `$2` must be strings to be concatenated, so they will be coerced if necessary.

In an assignment *v* ~=~ *e* or *v*~*op*=~*e*, the type of *v* becomes the type of *e*. In an ambiguous context like

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "`$1 == $2`" will succeed on any pair of the inputs

```
1    1.0    +1    0.1e+1    10E-1    001
```

but fail on the inputs

(null)	0
(null)	0.0
0a	0
1e50	1.0e50

There are two idioms for coercing an expression of one type to the other:

number ""	concatenate a null string to a *number* to coerce it to type string
string + 0	add zero to a *string* to coerce it to type numeric

Thus, to force a string comparison between two fields, say

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

Control Flow Statements

nawk provides **if–else, while, do–while,** and **for** statements, and statement grouping with braces, as in the C programming language.

The **if** statement syntax is

if (*expression*) $statement_1$ **else** $statement_2$

The *expression* acting as the conditional has no restrictions; it can include the relational operators **<, <=, >, >=, ==,** and **!=**; the regular expression matching operators ~ and !~ ; the logical operators || , **&&,** and **!**; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the *expression* is first evaluated. If it is non-zero and non-null, $statement_1$ is executed; otherwise $statement_2$ is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from "Arithmetic Functions" with an **if** statement results in

```
{     if (maxpop < $3) {
            maxpop = $3
            country = $1
      }
}
END   { print country, maxpop }
```

The **while** statement is exactly that of the C programming language:

while (*expression*) *statement*

The *expression* is evaluated; if it is non-zero and non-null the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```
{  i = 1
   while (i <= NF) {
      print $i
      i++
   }
}
```

The **for** statement is like that of the C programming language:

for (*expression*$_1$; *expression*; *expression*$_2$) *statement*

It has the same effect as

expression$_1$
while (*expression*) {
    statement
    expression$_2$
}

so

```
{ for (i = 1; i <= NF; i++)  print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the form

do *statement* **while** (*expression*)

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement* (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for**, which test for completion at the top of the loop.

The following example of a **do** statement prints all lines except those between `start` and `stop`.

```
/start/ {
            do {
                getline x
            } while (x !~ /stop/)
        }
        { print }
```

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **nawk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action, if any, is executed. Within the **END** action,

exit *expr*

causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero.

Arrays

nawk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array x. In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **nawk** program

```
    { x[NR] = $0 }
END { ... processing ... }
```

The first action merely records each input line in the array x, indexed by line number; processing is done in the **END** statement.

Array elements may also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array pop. The END action prints the total population of these two continents.

```
/Asia/     { pop["Asia"] += $3 }
/Africa/   { pop["Africa"] += $3 }
END        { print "Asian population in millions is", pop["Asia"]
             print "African population in millions is",
                 pop["Africa"] }
```

On the file **countries,** this program generates

```
Asian population in millions is 1765
African population in millions is 37
```

In this program if we had used pop[Asia] instead of pop["Asia"] the expression would have used the value of the variable Asia as the subscript, and since the variable is uninitialized, the values would have been accumulated in pop[""] .

Suppose our task is to determine the total area in each continent of the file **countries.** Any expression can be used as a subscript in an array reference. Thus

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array area and in that entry accumulates the value of the second field:

```
BEGIN       { FS = "\t" }
            { area[$4] += $2 }
END         { for (name in area)
                    print name, area[name] }
```

Invoked on the file **countries,** this program produces

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

for (*i* **in** *array*) *statement*

executes *statement* with the variable *i* set in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when *i* or *array* is altered during the loop.

nawk does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable **SUBSEP**). For example,

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
              arr[i,j] = ...
```

creates an array which behaves like a two-dimensional array; the subscript is the concatenation of *i*, **SUBSEP,** and *j*.

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating `area["Africa"]`, which would happen if we used

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array `area` contains an element with value `"Africa"`.

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The function

```
split("s1:s2:s3", a, ":")
```

splits the string `s1:s2:s3` into three fields, using the separator `:` , and stores `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]` . The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element may be deleted with the **delete** statement:

delete *arrayname* [*subscript*]

User-Defined Functions

nawk provides user-defined functions. A function is defined as

function *name*(*argument-list*) {
statements
}

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function (of course, using some input other than the file **countries**):

```
function fact(n) {
     if (n <= 1)
          return 1
     else
          return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables.) The **return** statement is optional, but the returned value is undefined if it is not included.

Some Lexical Conventions

Comments may be placed in **nawk** programs: they begin with the character # and end at the end of the line, as in

```
print x, y     # this is a comment
```

Statements in an **nawk** program normally occupy a single line. Several statements may occur on a single line if they are separated by semicolons. A long statement may be continued over several lines by terminating each continued line by a backslash. (It is not possible to continue a "..." string.) This explicit continuation is rarely necessary, however, since statements continue automatically if the line ends with a comma (for example, as might occur in a **print** or **printf** statement) or after the operators && and `||`.

Several pattern-action statements may appear on a single line if separated by semicolons.

Output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **nawk** lets you redirect output, so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement

print $expr_1, expr_2, \ldots, expr_n$

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0
```

To print an empty line use

```
print ""
```

Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN  { OFS = ":"; ORS = "\n\n" }
       { print $1, $2 }
```

Notice that

```
{ print $1 $2 }
```

prints the first and second fields with no intervening output field separator, because `$1 $2` is a string consisting of the concatenation of the first two fields.

The printf Statement

nawk's printf statement is the same as that in C except that the * format specifier is not supported. The **printf** statement has the general form

printf *format*, $expr_1$, $expr_2$, . . ., $expr_n$

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Figure 4-8. Each specification begins with a %, ends with a letter that determines the conversion, and may include

-	left-justify expression in its field
width	pad field to this width as needed; fields that begin with a leading 0 are padded with zeros
.prec	maximum string width or digits to right of decimal point

Character	Prints Expression as
c	single character
d	decimal number
e	[–]**d.ddddddE**[+–]**dd**
f	[–]**ddd.dddddd**
g	**e** or **f** conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

Figure 4-8: **nawk printf** Conversion Characters

Here are some examples of **printf** statements along with the corresponding output:

```
printf "%d", 99/2                      49
printf "%e", 99/2                      4.950000e+01
printf "%f", 99/2                      49.500000
printf "%6.2f", 99/2                   49.50
printf "%g", 99/2                      49.5
printf "%o", 99                        143
printf "%06o", 99                      000143
printf "%x", 99                        63
printf "|%s|", "January"               |January|
printf "|%10s|", "January"             |   January|
printf "|%-10s|", "January"            |January   |
printf "|%.3s|", "January"             |Jan|
printf "|%10.3s|", "January"           |       Jan|
printf "|%-10.3s|", "January"          |Jan       |
printf "%%"                            %
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into Files

It is possible to print output into files instead of to the standard output by using the > and >> redirection operators. For example, the following program invoked on the file **countries** prints all lines where the population (third field) is bigger than 100 into a file called `bigpop`, and all other lines into `smallpop`:

```
$3 > 100    { print $1, $3 >"bigpop" }
$3 <= 100   { print $1, $3 >"smallpop" }
```

Notice that the file names have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. If the output file names were created by an expression, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the > operator has higher precedence than concatenation; without parentheses, the concatenation of `tmp` and `FILENAME` would not work.

NOTE

Files are opened once in an **nawk** program. If > is used to open a file, its original contents are overwritten. But if >> is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of into a file. The statement

print | "*command-line*"

causes the output of **print** to be piped into the *command-line*.

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and file names can come from variables and the return values from functions, for instance.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The **nawk** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called pop. Then it prints each continent and its population, and pipes this output into the sort command.

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (c in pop)
              print c ":" pop[c] | "sort" }
```

Invoked on the file **countries,** this program yields

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named sort), but they are created and opened only once in the entire run. So, in the last example, for all c in pop, only one sort pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement **close**(*file*) closes a file or pipe; *file* is the string used to create it in the first place, as in

```
close("sort")
```

When opening or closing a file, different strings are different commands.

Input

The most common way to give input to an **nawk** program is to name on the command line the file(s) that contains the input. This is the method we've been using in this chapter. However, there are several other methods we could use, each of which this section describes.

Files and Pipes

You can provide input to an **nawk** program by putting the input data into a file, say **nawkdata,** and then executing

nawk '*program*' nawkdata

nawk reads its standard input if no file names are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **nawk.** For example, **egrep**(1) selects input lines containing a specified regular expression, but it can do so faster than **nawk** since this is the only thing it does. We could, therefore, invoke the pipe

egrep 'Asia' countries | nawk '...'

egrep quickly finds the lines containing **Asia** and passes them on to the **nawk** program for subsequent processing.

Input Separators

With the default setting of the field separator **FS,** input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
      field1   field2
   field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS.** For example,

```
BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" }
```

sets it to an optional comma followed by any number of blanks and tabs. **FS** can also be set on the command line with the **–F** argument:

nawk –**F** '(,[\t]*) | ([\t]+)' '...'

behaves the same as the previous example. Regular expressions used as field separators match the left-most longest occurrences (as in **sub()**), but do not match null strings.

Multi-line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed, though only in a limited way. If the built-in record separator variable **RS** is set to the empty string, as in

```
BEGIN    { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to use

```
BEGIN    { RS = ""; FS = "\n" }
```

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. "The **getline** Function" and "Cooperation with the Shell" in this chapter show other examples of processing multi-line records.

The getline Function

nawk's facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting **RS** to null doesn't work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function **getline** can be used to read input either from the current input or from a file or pipe, by redirection analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF, NR,** and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and –1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with START and ends with a line beginning with STOP. The following **nawk** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

```
f[1] f[2] ... f[nf]
```

Once the line containing STOP is encountered, the record can be processed from the

data in the `f array`:

```
/^START/ {
            f[nf=1] = $0
            while (getline && $0 !~ /^STOP/)
                 f[++nf] = $0
            # now process the data in f[1]...f[nf]
            ...
}
```

Notice that this code uses the fact that **&&** evaluates its operands left to right and stops as soon as one is true.

The same job can also be done by the following program:

```
/^START/ && nf==0     { f[nf=1] = $0 }
nf > 1                { f[++nf] = $0 }
/^STOP/               { # now process the data in f[1]...f[nf]
                        ...
                        nf = 0
}
```

The statement

```
getline x
```

reads the next record into the variable `x`. No splitting is done; **NF** is not set. The statement

```
getline <"file"
```

reads from `file` instead of the current input. It has no effect on **NR** or **FNR,** but field splitting is performed and **NF** is set. The statement

```
getline x <"file"
```

gets the next record from `file` into `x`; no splitting is done, and **NF, NR** and **FNR** are untouched.

NOTE

If a filename is an expression, it needs to be placed in parentheses for correct evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) { ... }
```

This is because the < has precedence over concatenation. Without parentheses, a statement such as

```
getline x < "tmp" FILENAME
```

sets `x` to read the file `tmp` and not `tmp` *<value of FILENAME>*.

Also, if you use this **getline** statement form, a statement like

```
while ( getline x < file ) { ... }
```

loops forever if the file cannot be read, because `getline` returns −1, not zero, if an error occurs. A better way to write this test is

```
while ( getline x < file > 0) { ... }
```

It is also possible to pipe the output of another command directly into **getline.** For example, the statement

```
while ("who" | getline)
    n++
```

executes `who` and pipes its output into `getline`. Each iteration of the `while` loop reads one more line and increments the variable `n`, so after the `while` loop terminates, `n` contains a count of the number of users. Similarly, the statement

```
"date" | getline d
```

pipes the output of `date` into the variable `d`, thus setting `d` to the current date. Figure 4-9 summarizes the **getline** function.

Form	Sets
getline	**$0, NF, NR, FNR**
getline *var*	*var*, **NR, FNR**
getline <*file*	**$0, NF**
getline *var* <*file*	*var*
cmd | **getline**	**$0, NF**
cmd | **getline** *var*	*var*

Figure 4-9: **getline** Function

Command-line Arguments

The command-line arguments are available to an **nawk** program: the array **ARGV** contains the elements **ARGV[0]**, ..., **ARGV[ARGC−1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **nawk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments). The following command line contains an **nawk** program that echoes the arguments that appear after the program name:

```
nawk '
BEGIN {
  for (i = 1; i < ARGC; i++)
      printf "%s ", ARGV[i]
  printf "\n"
}' $*
```

The arguments may be modified or added to; **ARGC** may be altered. As each input file ends, **nawk** treats the next non-null element of **ARGV** (up to the current value of **ARGC–1**) as the name of the next input file.

There is one exception to the rule that an argument is a file name: if it is of the form

var=value

then the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a file name. If *value* is a string, no quotes are needed.

Using nawk with Other Commands and the Shell

nawk gains its greatest power when it is used in conjunction with other programs. Here we describe some of the ways in which **nawk** programs cooperate with other commands.

The system Function

The built-in function **system**(*command-line*) executes the command *command-line*, which may well be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed.

For example, the program

```
$1 == "#include"  { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

calls the command `cat` to print the file named in the second field of every input record whose first field is `#include`, after stripping any <, > or " that might be present.

Cooperation with the Shell

In all the examples thus far, the **nawk** program was in a file and fetched from there using the **–f** flag, or it appeared on the command line enclosed in single quotes, as in

nawk '{ print $1 }' ...

Since **nawk** uses many of the same characters as the shell does, such as $ and ", surrounding the **nawk** program with single quotes ensures that the shell will pass the entire program unchanged to the **nawk** interpreter.

Now, consider writing a command **addr** that will search a file **addresslist** for name, address and telephone information. Suppose that **addresslist** contains names and addresses in which a typical entry is a multi-line record such as

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

We want to search the address list by issuing commands like

addr Emlin

That is easily done by a program of the form

```
nawk '
BEGIN           { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called **addr** that contains

```
nawk '
BEGIN           { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **nawk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The `$1` is outside the quotes, visible to the shell, which therefore replaces it by the pattern `Emlin` when the command **addr Emlin** is invoked. On a UNIX system, **addr** can be made executable by changing its mode with the following command: **chmod +x addr**.

A second way to implement **addr** relies on the fact that the shell substitutes for **$** parameters within double quotes:

```
nawk "
BEGIN           { RS = \"\" }
/$1/
" addresslist
```

Here we must protect the quotes defining `RS` with backslashes, so that the shell passes them on to **nawk,** uninterpreted by the shell. `$1` is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr** *pattern* is invoked.

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **nawk** program that explicitly reads through the address list with **getline:**

```
nawk '
BEGIN   { RS = ""
          while (getline < "addresslist")
             if ($0 ~ ARGV[1])
                print $0
} ' $*
```

All processing is done in the BEGIN action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

Example Applications

nawk can be used in surprising ways. We have seen **nawk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **nawk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. In this section, we will present a few more examples to illustrate some additional **nawk** programs.

Generating Reports

nawk is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file **countries** in which we list the continents alphabetically, and after each continent its countries in decreasing order of population:

```
Africa:
                    Sudan          19
                    Algeria        18

Asia:
                    China         866
                    India         637
                    USSR          262

Australia:
                    Australia      14

North America:
                    USA           219
                    Canada         24

South America:
                    Brazil        116
                    Argentina      26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, we create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program **triples,** which uses an array pop indexed by subscripts of the form 'continent:country' to store the population of a given country. The print statement in the END section of the program creates the list of continent-country-population

triples that are piped to the `sort` routine.

```
BEGIN   { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END     { for (cc in pop)
              print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for `sort` deserve special mention. The `-t:` argument tells `sort` to use `:` as its field separator. The `+0 -1` arguments make the first field the primary sort key. In general, *+i −j* makes fields *i+1*, *i+2*, ..., *j* the sort key. If *−j* is omitted, the fields from *i+1* to the end of the record are used. The `+2nr` argument makes the third field, numerically decreasing, the secondary sort key (n is for numeric, `r` for reverse order). Invoked on the file **countries,** this program produces as output

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form we run it through a second **nawk** program **format**:

```
BEGIN   { FS = ":" }
{       if ($1 != prev) {
             print "\n" $1 ":"
             prev = $1
        }
        printf "\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

nawk −f triples countries | nawk −f format

gives us our desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **nawks** and **sorts.**

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

Word Frequencies

Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```
    { for (w = 1; w <= NF; w++) count[$w]++ }
END { for (w in count) print count[w], w | "sort -nr" }
```

The first statement uses the array `count` to accumulate the number of times each word is used. Once the input has been read, the second `for` loop pipes the final count along with each word into the `sort` command.

Accumulation

Suppose we have two files, `deposits` and `withdrawals`, of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
nawk '
FILENAME == "deposits"      { balance[$1] += $2 }
FILENAME == "withdrawals"   { balance[$1] -= $2 }
END                         { for (name in balance)
                                  print name, balance[name]
} ' deposits withdrawals
```

The first statement uses the array `balance` to accumulate the total amount for each name in the file `deposits`. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name will be created by the second statement. The `END` action prints each name with its net balance.

Random Choice

The following function prints (in order) `k` random elements from the first `n` elements of the array `A`. In the program, `k` is the number of entries that still need to be printed, and `n` is the number of elements yet to be examined. The decision of whether to print the *i*th element is determined by the test `rand() < k/n`.

```
function choose(A, k, n) {
        for (i = 1; n > 0; i++)
                if (rand() < k/n--) {
                        print A[i]
                        k--
                }
        }
}
```

Shell Facility

The following **nawk** program simulates (crudely) the history facility of the UNIX system shell. A line containing only = re-executes the last command executed. A line beginning with = *cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
                system(x[NR] = x[NR-1])
            else
                for (i = NR-1; i > 0; i--)
                        if (x[i] ~ $2) {
                                system(x[NR] = x[i])
                                break
                        }
            next }

/./         { system(x[NR] = $0) }
```

Form-letter Generation

The following program generates form letters, using a template stored in a file called `form.letter`:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

and replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The BEGIN action stores the template in the array `template;` the remaining action cycles through the input data, using `gsub` to replace template fields of the form `$n` with the corresponding data fields.

```
BEGIN {         FS = "|"
                while (getline <"form.letter")
                 line[++n] = $0
}
{         for (i = 1; i <= n; i++) {
                 s = line[i]
                 for (j = 1; j <= NF; j++)
                         gsub("\\$"j, $j, s)
                 print s
          }
}
```

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

nawk Summary

Command Line

nawk *program filenames*
nawk **–f** *program-file filenames*
nawk **–F***s* sets field separator to string *s*; **–Ft** sets separator to tab

A program consists of a sequence of pattern-action and function definitions which are separated by either newlines or semicolons.

Patterns

BEGIN
END
/regular expression/
relational expression
pattern && *pattern*
pattern | | *pattern*
(*pattern*)
!*pattern*
pattern, *pattern*

Control Flow Statements

if (*expr*) *statement* [else *statement*]
if (*subscript* **in** *array*) *statement* [else *statement*]
while (*expr*) *statement*
for (*expr*; *expr*; *expr*) *statement*
for (*var* **in** *array*) *statement*
do *statement* **while** (*expr*)
break
continue
next
exit [*expr*]
return [*expr*]

Input-output

close(*filename***)**	close file
getline	set **$0** from next input record; set **NF, NR, FNR**
getline <*file*	set **$0** from next record of *file*; set **NF**
getline *var*	set *var* from next input record; set **NR, FNR**
getline *var* <*file*	set *var* from next record of *file*
print	print current record
print *expr-list*	print expressions
print *expr-list* >*file*	print expressions on *file*
printf *fmt*, *expr-list*	format and print
printf *fmt*, *expr-list* >*file*	format and print on *file*
system(*cmd-line***)**	execute command *cmd-line*, return status

In **print** and **printf** above, >>*file* appends to the *file*, and | *command* writes on a pipe. Similarly, *command* | **getline** pipes into **getline**. **getline** returns 0 on end of file, and −1 on error.

Functions

func *name*(*parameter list*) { *statement* }
function *name*(*parameter list*) { *statement* }
function-name(*expr*, *expr*, ...)

String Functions

gsub(*r*,*s*,*t*)	substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use **$0**
index(*s*,*t*)	return index of string *t* in string *s*, or 0 if not present
length(*s*)	return length of string *s*
match(*s*, *r*)	return position in *s* where regular expression *r* occurs, or 0 if *r* is not present
split(*s*,*a*,*r*)	split string *s* into array *a* on regular expression *r*, return number of fields; if *r* omitted, **FS** is used in its place
sprintf(*fmt*, *expr-list*)	print *expr-list* according to *fmt*, return resulting string
sub(*r*,*s*,*t*)	like gsub except only the first matching substring is replaced
substr(*s*,*i*,*n*)	return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

Arithmetic Functions

atan2(*y*,*x*)	arctangent of *y* / *x* in radians
cos(*expr*)	cosine (angle in radians)
exp(*expr*)	exponential
int(*expr*)	truncate to integer
log(*expr*)	natural logarithm
rand()	random number between 0 and 1
sin(*expr*)	sine (angle in radians)
sqrt(*expr*)	square root
srand(*expr*)	new seed for random number generator; use time of day if no *expr*

Operators (Increasing Precedence)

`= += -= *= /= %= ^=`	assignment
`?:`	conditional expression
`||`	logical OR
`&&`	logical AND
`~ !~`	regular expression match, negated match
`< <= > >= != ==`	relationals
blank	string concatenation
`+ -`	add, subtract
`* / %`	multiply, divide, mod
`+ - !`	unary plus, unary minus, logical negation
`^`	exponentiation (** is a synonym)
`++ --`	increment, decrement (prefix and postfix)
`$`	field

Regular Expressions (Increasing Precedence)

c	matches non-metacharacter *c*
c	matches literal character *c*
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[*abc...*]	character class matches any of *abc...*
[^*abc...*]	negated class matches any but *abc...* and newline
r1 | *r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r+	matches one or more *r*'s
*r**	matches zero or more *r*'s
r?	matches zero or one *r*
(*r*)	grouping: matches *r*

Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0..ARGC-1)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default %.6g)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	index of first character matched by **match()**; 0 if no match
RLENGTH	length of string matched by **match()**; –1 if no match
SUBSEP	separates multiple subscripts in array elements; default "\034"

Limits

Any particular implementation of **nawk** enforces some limits. Here are typical values:

100 fields
3072 characters per input record
3072 characters per output record
3072 characters per individual field
3072 characters per **printf** string
400 characters maximum quoted string
256 characters in character class
open files determined by IRIX™ kernel (see **getdtablesize**(3))
1 pipe
100 function nesting levels
50 associative arrays

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

var = *expr*

its type is set to that of the expression. (Assignment includes +=, −=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

(that is, concatenation with a null string).

Uninitialized variables have the numeric value **0** and the string value "". Accordingly, if `x` is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But the following is false:

```
if (x == "0") ...
```

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that `$1` is to be numeric, and

```
$1 = $1 "," $2
```

implies that `$1` and `$2` are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value "" ; they are not numeric. Nonexistent fields (i.e., fields past **NF**) are treated this way, too.

As it is for fields, so it is for array elements created by **split()**.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if `arr[i]` does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" so the **if** is satisfied. The special construction

```
if (i in arr) ...
```

determines if `arr[i]` exists without the side effect of creating it if it does not.

An Overview of awk Programming

Suppose you want to tabulate some survey results stored in a file; print various reports summarizing these results; generate form letters; reformat a data file for one application package to use with another package; count the occurrences of particular strings in a file; or globally substitute a string in many files without ever invoking an editor. The tasks common to all these examples are retrieving and processing data in files. **awk** simplifies these tasks. **awk** (the name is an acronym constructed from the initials of its developers) is an interpretive programming language designed to handle these tasks. It is also the name of the UNIX system command you use to run **awk** programs. This tutorial describes the language and command.

As these examples suggest, **awk** is suited for various applications. It is designed to retrieve and manipulate data efficiently from any files containing mixtures of words and/or numbers. **awk** has been most commonly used to generate reports. However, **awk** also has been used to choose names randomly, to sort error messages in a programmer's manual, and even to implement database systems and compilers.

Why should you choose **awk** instead of another language to implement these tasks? **awk** is a relatively easy language to learn. Its syntax is simple, but powerful, and can be summarized in a few pages. For instance, unlike C, Pascal, and some other languages, **awk** does not require that you explicitly initialize variables in programs. However, it provides the same powerful control-flow statements (like **for** and **if-else**) provided by these other languages. **awk** can also save you some time, because you don't need to compile **awk** programs before running them. This makes it a good language for prototyping.

This chapter begins with the basics of **awk**, and is intended to make it easy for you to start writing and running your own **awk** programs. The rest of the chapter describes the complete language and is somewhat less tutorial. For the experienced **awk** user, there's a summary of the language at the end of the chapter.

You should be familiar with the UNIX system and shell programming to use this chapter. Although you don't need other programming experience, some knowledge of C is beneficial, because many constructs found in **awk** are also found in C.

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Program Structure

The basic operation of **awk**(1) is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern action statements, as Figure 4-1 shows.

Structure:

pattern { *action* }
pattern { *action* }
...

Example:

```
$1 == "address"  { print $2, $3 }
```

Figure 4-1: **awk** Program Structure and Example

The example in the figure is a typical **awk** program, consisting of one pattern-action statement. Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in

```
$1 == "name"
```

the matching line is printed. If there is no pattern with an action, as in

```
{ print $1, $2 }
```

then the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Using awk

There are two ways to run an **awk** program. First, you can type the command line

awk '*pattern-action statements*' *optional list of input files*

to execute the pattern-action statements on the set of named input files. For example, you could say

awk '{ print $1, $2 }' file1 file2

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk**(1) reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (–) as one of the input files. For example,

awk '{ print $3, $4 }' file1 –

says to read input first from **file1** and then from the standard input.

The arrangement above is convenient when the **awk** program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file and use the **–f** option to fetch it:

awk –f *programfile* *optional list of input files*

For example, the following command line says to fetch **myprogram** and read from the file **file1**:

awk –f myprogram file1

Fields

awk(1) normally reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline. **awk** then splits each record into fields, where, by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the following file, **countries**. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent where it is, for the ten largest countries in the world. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The wide space between fields is a tab in the original input; a single blank separates `North` and `South` from `America`.

```
USSR        8650    262    Asia
Canada      3852    24     North America
China       3692    866    Asia
USA         3615    219    North America
Brazil      3286    116    South America
Australia   2968    14     Australia
India       1269    637    Asia
Argentina   1072    26     South America
Sudan       968     19     Africa
Algeria     920     18     Africa
```

Figure 4-2: The Sample Input File **countries**

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, in which case the first record of **countries** would have four fields, the second five, and so on. It's possible to set the field separator to just tab, so each line would have four fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs.

The first field within a line is called **$1**, the second **$2**, and so forth. An entire record is called **$0**.

Printing

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line; you can accomplish this with an **awk** program consisting of a single **print** statement

```
{ print }
```

so the command line

awk '{ print }' countries

prints each line of **countries**, copying the file to the standard output. The **.print** statement can also be used to print parts of a record; for instance, the program

```
{ print $1, $3 }
```

prints the first and third fields of each record. Thus

awk '{ print $1, $3 }' countries

produces as output the sequence of lines:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator, which by default is a single blank. Each line printed is terminated by the output record separator, which by default is a newline.

NOTE

In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. Each complete program can be run either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the **–f** flag, as discussed in "**awk** Command Usage." In an example, if no input is mentioned, it is assumed to be the file **countries**.

Formatted Printing

For more carefully formatted output, **awk** provides a C-like **printf** statement

printf *format*, $expr_1$, $expr_2$, . . ., $expr_n$

which prints the $expr_i$'s according to the specification in the string *format*. For example, the **awk** program

```
{ printf "%10s %6d\n", $1, $3 }
```

prints the first field ($1) as a string of 10 characters (right justified), then a space, then the third field ($3) as a decimal number in a six-character field, then a newline (\n). With input from the file **countries,** this program prints an aligned table:

```
      USSR   262
    Canada    24
     China   866
       USA   219
    Brazil   116
 Australia    14
     India   637
 Argentina    26
     Sudan    19
   Algeria    18
```

With **printf,** no output separators or newlines are produced automatically; you must create them yourself, which is the purpose of the **\n** in the format specification. "The **printf** Statement" in this chapter contains a full description of **printf**.

Simple Patterns

You can select specific records for printing or other processing with simple patterns. **awk** has three kinds of patterns that you can use. First, you can use patterns called relational expressions that make comparisons. For example, the operator == tests for equality. To print the lines for which the fourth field equals the string Asia, we can use the program consisting of the single pattern

```
$4 == "Asia"
```

With the file **countries** as input, this program yields

```
USSR        8650     262     Asia
China       3692     866     Asia
India       1269     637     Asia
```

The complete set of comparisons is >, >=, <, <=, == (equal to) and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with more than 100 million population. The program

```
$3 > 100
```

is all that is needed (remember that the third field in the file **countries** is the population in millions); it prints all lines in which the third field exceeds 100.

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters US anywhere; with the file **countries** as input, it prints

```
USSR        8650    262     Asia
USA         3615    219     North America
```

We will have a lot more to say about regular expressions later in this chapter.

Third, you can use two special patterns, **BEGIN** and **END,** that match before the first record has been read and after the last record has been processed. This program uses **BEGIN** to print a title:

```
BEGIN       { print "Countries of Asia:" }
/Asia/      { print "     ", $1 }
```

The output is

```
Countries of Asia:
     USSR
     China
     India
```

Simple Actions

We have already seen the simplest action of an **awk** program: printing each input line. Now let's consider how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in Variables

Besides reading the input and splitting it into fields, **awk**(1) counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields. So the program

```
{ print NR, NF }
```

prints the number of each line and how many fields it has, while

```
{ print NR, $0 }
```

prints each record preceded by its record number.

User-defined Variables

In addition to the built-in variables like **NF** and **NR, awk** lets you define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file **countries**:

```
        { sum = sum + $3 }
END     { print "Total population is", sum, "million"
          print "Average population of", NR, "countries is", sum/NR }
```

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

awk has built-in functions that handle common arithmetic and string operations for you. For example, there's an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. **awk** also lets you define your own functions. Functions are described in detail under "Actions" in this chapter.

Useful One-Line awk Programs

Although **awk** can be used to write large programs of some complexity, many programs are not much more complicated than what we've seen so far. Here is a collection of other short programs that you might find useful and instructive. They are not explained here, but any new constructs do appear later in this chapter.

Print last field of each input line:

```
{ print $NF }
```

Print 10th input line:

```
NR == 10
```

Print last input line:

```
{ line = $0}
END{ print line }
```

Print input lines that don't have 4 fields:

```
NF != 4 { print $0, " does not have 4 fields" }
```

Print input lines with more than 4 fields:

```
NF > 4
```

Print input lines with last field more than 4:

```
$NF > 4
```

Print total number of input lines:

```
END{ print NR }
```

Print total number of fields:

```
{ nf = nf + NF }
END{ print nf }
```

Print total number of input characters:

```
{ nc = nc + length($0) }
END{ print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
/Asia/{ nlines++ }
END{ print nlines }
```

(The statement `nlines++` has the same effect as `nlines = nlines + 1`.)

Error Messages

If you make an error in your **awk** program, you generally get a message like

```
awk: syntax error near source line 2
awk: bailing out near source line 2
```

The first message means that you have made a grammatical error that was finally detected near the line specified. The second message means that because of the syntax errors **awk**(1) made no attempt to execute your program.

Sometimes you get a little more help about what the error is, for instance a report of missing braces or unbalanced parentheses. For example, running the program

```
$3 < 200 { print $1, $3
```

which is missing a closing brace, generates the error messages

```
awk: syntax error near line 2
awk: illegal statement near line 2
```

Some errors may be detected when your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (**NR**) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **awk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once before the **awk** command starts to read its first input record. The pattern **END** matches the end of the input, after the last record has been processed.

The following **awk** program uses **BEGIN** to set the field separator to tab (**t**) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can be continued after a comma.)

```
BEGIN { FS = "\t"
        printf "%10s %6s %5s   %s\n",
         "COUNTRY", "AREA", "POP", "CONTINENT" }
      { printf "%10s %6d %5d   %s\n", $1, $2, $3, $4
         area = area + $2; pop = pop + $3 }
END   { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file **countries** as input, this program produces

```
COUNTRY    AREA   POP   CONTINENT
   USSR    8650   262   Asia
 Canada    3852    24   North America
  China    3692   866   Asia
    USA    3615   219   North America
 Brazil    3286   116   South America
Australia  2968    14   Australia
  India    1269   637   Asia
Argentina  1072    26   South America
  Sudan     968    19   Africa
Algeria     920    18   Africa

  TOTAL   30292  2201
```

Relational Expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. **awk** has six relational operators, and two regular expression matching operators, ~ (tilde) and !~, which are discussed in the next section, for making comparisons. Figure 4-3 shows these operators and their meanings.

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

Figure 4-3: **awk** Comparison Operators

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can tell what is intended. The full story is in "Number or String?" in this chapter.) Thus, the pattern `$3>100` selects lines where the third field exceeds 100, and the program

```
$1 >= "S"
```

selects lines that begin with the letters S, T, U, through Z, which are

```
USA         3615    219    North America
Sudan       968     19     Africa
```

In the absence of any other information, **awk** treats fields as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters, and with the file **countries** as input, prints the single line for which this test succeeds:

```
Australia    2968    14    Australia
```

If both fields appear to be numbers, the comparisons are done numerically.

Regular Expressions

awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called regular expressions, and are like those in **egrep**(1) and **lex**(1). The simplest regular expression is a string of characters enclosed in slashes, like

```
/Asia/
```

This program prints all input records that contain any occurrence of `Asia`. (If a record contains `Asia` as part of a larger string like `Asian` or `Pan-Asiatic`, it is also printed.) In general, if *re* is a regular expression, then the pattern

/re/

matches any line that contains a substring specified by the regular expression *re*.

To restrict a match to a specific field, you use the matching operators ~ (for matches) and **!~** (for does not match). The program

```
$4 ~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field matches `Asia`, while the program

```
$4 !~ /Asia/ { print $1 }
```

prints the first field of all lines in which the fourth field does not match `Asia`.

In regular expressions, the symbols

\ ^ $. [] * + ? () |

are metacharacters with special meanings like the metacharacters in the UNIX shell. For example, the metacharacters ^ and $ match the beginning and end, respectively, of a string, and the metacharacter . matches any single character. Thus,

```
/^.$/
```

matches all records that contain exactly one character.

A group of characters enclosed in brackets matches any one of the enclosed characters; for example, /[**ABC**]/ matches records containing any one of **A, B**, or **C** anywhere. Ranges of letters or digits can be abbreviated within brackets: /[**a-zA-Z**]/ matches any single letter.

If the first character after the [is a **^**, this complements the class so it matches any character not in the set: .B /[^a-zA-Z]/ matches any non-letter. The program

```
$2 !~ /^[0-9]+$/
```

prints all records in which the second field is not a string of one or more digits (^ for beginning of string, `[0-9]+` for one or more digits, and `$` for end of string). Programs of this nature are often used for data validation.

Parentheses () are used for grouping and the pipe symbol | is used for alternatives. The program

```
/(apple|cherry) (pie|tart)/
```

matches lines containing any one of the four substrings `apple pie`, `apple tart`, `cherry pie`, or `cherry tart` .

To turn off the special meaning of a metacharacter, precede it by a \ (backslash). Thus, the program

```
/a\$/
```

prints all lines containing a followed by a dollar sign.

In addition to recognizing metacharacters, the **awk** command recognizes the following C escape sequences within regular expressions and strings:

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
ddd	octal value *ddd*
\"	quotation mark
c	any other character *c* literally

For example, to print all lines containing a tab, use the program

```
/\t/
```

awk interprets any string or variable on the right side of a ~ or !~ as a regular expression. For example, we could have written program

```
$2 !~ /^[0-9]+$/
```

as

```
BEGIN     { digits = "^[0-9]+$" }
$2 !~ digits
```

When a literal quoted string like **"^[0-9]+$** is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. The reason may seem arcane, but it is merely that one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing the letter A followed by a dollar sign. The regular expression for this pattern is **a\$**. If we want to create a string to represent this regular expression, we must add one more backslash: **"a\\$"**. The regular expressions on each of the following lines are equivalent:

```
x ~ "a\\$"          x ~ /a\$/
x ~ "a\$"           x ~ /a$/
x ~ "a$"            x ~ /a$/
x ~ "\\t"           x ~ /\t/
```

Of course, if the context of a matching operator is

```
x ~ $1
```

then the additional level of backslashes is not needed in the first field.

The precise form of regular expressions and the substrings they match is given in Figure 4-4. The unary operators *, +, and ? have the highest precedence, then concatenation, and then alternation |. All operators are left associative.

Expression	Matches
c	any non-metacharacter *c*
c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
$r_1 r_2$	r_1 then r_2 (concatenation)
$r_1 \vert r_2$	r_1 or r_2 (alternation)

Figure 4-4: **awk** Regular Expressions

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (or), && (and), and ! (not). For example, suppose we want to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is `Asia` and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The program

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with `Asia` or `Africa` as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator |:

```
$4 ~ /^(Asia|Africa)$/
```

The negation operator **!** has the highest precedence, then **&&**, and finally ||. The operators **&&** and || evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in

pat_1, pat_2 { ... }

In this case, the action is performed for each line between an occurrence of pat_1 and the next occurrence of pat_2 (inclusive). As an example, the pattern

```
/Canada/, /Brazil/
```

matches lines starting with the first line that contains the string Canada up through the next occurrence of the string Brazil :

```
Canada      3852    24    North America
China       3692    866   Asia
USA         3615    219   North America
Brazil      3286    116   South America
```

Similarly, since **FNR** is the number of the current record in the current input file (and **FILENAME** is the name of the current input file), the program

```
FNR == 1, FNR == 5 { print FILENAME, $0 }
```

prints the first five records of each input file with the name of the current input file prepended.

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they may also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in Variables

Figure 4-5 lists the built-in variables that **awk** maintains. Some of these we have already met; others are used in this and later sections.

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	set by **match()**	-
RLENGTH	set by **match()**	-

Figure 4-5: **awk** Built-in Variables

Arithmetic

Actions use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file **countries**. Since the second field is the area in thousands of square miles and the third field is the population in millions, the expression **1000 * $3 / $2** gives the population density in people per square mile. The program

```
{ printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

applied to the file **countries** prints the name of each country and its population density:

```
     USSR   30.3
   Canada    6.2
    China  234.6
      USA   60.6
   Brazil   35.3
Australia    4.7
    India  502.0
Argentina   24.3
    Sudan   19.6
  Algeria   19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are **+**, **–**, *****, **/**, **%** (remainder) and **^** (exponentiation; ****** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **awk** recognizes and produces scientific (exponential) notation: `1e6`, `1E6`, `10e5`, and `1000000` are numerically equal.

awk has assignment statements like those found in C. The simplest form is the assignment statement

$v = e$

where v is a variable or field name, and e is an expression. For example, to compute the total population and number of Asian countries, we could write

```
$4 == "Asia"        { pop = pop + $3; n = n + 1 }
END                 { print "population of", n,\
                    "Asian countries in millions is", pop }
```

A long **awk** statement can also be split across several lines by continuing each line with a \, as in the **END** action shown here. Applied to **countries**, this program produces

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern `$4 == "Asia"` contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **awk** initializes each variable with the string value "" and the numeric value **0**.

The assignments in the previous program can be written more concisely using the operators += and ++:

```
$4 == "Asia"        { pop += $3; ++n }
```

The operator += is borrowed from C. It has the same effect as the longer version — the variable on the left is incremented by the value of the expression on the right — but += is shorter and runs faster. The same is true of the ++ operator, which adds one to a variable.

The abbreviated assignment operators are +=, −=, *=, /=, %=, and ^=. Their meanings are similar:

v op= e

has the same effect as

v = v op e.

The increment operators are ++ and —. As in C, they may be used as prefix operators (**++x**) or postfix (**x++**). If **x** is 1, then **i=++x** increments **x,** then sets **i** to 2, while **i=x++** sets **i** to 1, then increments **x**. An analogous interpretation applies to prefix and postfix —.

Assignment and increment and decrement operators may all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3         { maxpop = $3; country = $1 }
END                 { print country, maxpop }
```

Note, however, that this program would not be correct if all values of $3 were negative.

awk provides the built-in arithmetic functions shown in Figure 4-6.

Function	Value Returned
atan2(y,x)	arctangent of y/x in the range $-\pi$ to π
cos(x)	cosine of x, with x in radians
exp(x)	exponential function of x
int(x)	integer part of x truncated towards 0
log(x)	natural logarithm of x
rand()	random number between 0 and 1
sin(x)	sine of x, with x in radians
sqrt(x)	square root of x
srand(x)	x is new seed for rand()

Figure 4-6: **awk** Built-in Arithmetic Functions

x and y are arbitrary expressions. The function **rand**() returns a pseudo-random floating point number in the range (0,1), and **srand**(x) can be used to set the seed of the generator. If **srand**() has no argument, the seed is derived from the time of day.

Strings and String Functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in **"abc"** or **"hello, everyone"** . String constants may contain the C escape sequences for special characters listed in "Regular Expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The program

```
{ print NR ":" $0 }
```

prints each record preceded by its record number and a colon, with no blanks. The three strings representing the record number, the colon, and the record are concatenated and the resulting string is printed. The concatenation operator has no explicit representation other than juxtaposition.

awk provides the built-in string functions shown in Figure 4-7. In this table, *r* represents a regular expression (either as a string or as /*r*/), *s* and *t* string expressions, and *n* and *p* integers.

Function	Description
gsub(*r*,*s*)	substitute *s* for *r* globally in current record, return number of substitutions
gsub(*r*,*s*,*t*)	substitute *s* for *r* globally in string *t*, return number of substitutions
index(*s*,*t*)	return position of string *t* in *s*, 0 if not present
length	return length of **$0**
length(*s*)	return length of *s*
match(*s*, *r*)	return the position in *s* where *r* occurs
split(*s*,*a*)	split *s* into array *a* on FS, return number of fields
split(*s*,*a*,*r*)	split *s* into array *a* on *r*, return number of fileds
sprintf(*fmt*,*expr -list*)	return *expr -list* formatted according to format string *fmt*
sub(*r*,*s*)	substitute *s* for first *r* in current record, return number of substitutions
sub(*r*,*s*,*t*)	substitute *s* for first *r* in *t*, return number of substitutions
substr(*s*,*p*)	return suffix of *s* starting at position *p*
substr(*s*,*p*,*n*)	return substring of *s* of length *n* starting at position *p*

Figure 4-7: **awk** Built-in String Functions

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(1) . The function **gsub**(*r*,*s*,*t*) replaces successive occurrences of substrings matched by the regular expression *r* with the replacement string *s* in the target string *t*. (As in **ed,** leftmost longest matches are used.) It returns the number of substitutions made. The function **gsub**(*r*,*s*) is a synonym for **gsub**(*r*,*s*,**$0**) . For example, the program

```
{ gsub(/USA/, "United States"); print }
```

transcribes its input, replacing occurrences of `USA` by `United States`. The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index**(*s*,*t*) returns the leftmost position where the string *t* begins in *s*, or zero if *t* does not occur in *s* . The first character in a string is at position 1. For example,

```
index("banana", "an")
```

returns 2.

The **length** function returns the number of characters in its argument string; thus,

```
{ print length($0), $0 }
```

prints each record, preceded by its length. (**$0** does not include the input record separator.) The program

```
length($1) > max  { max = length($1); name = $1 }
END                           { print name }
```

applied to the file **countries** prints the longest country name: `Australia`.

The **match**(*s, r*) function returns the position in string *s* where regular expression *r* occurs, or 0 if it does not occur. This function makes use of the two built-in variables **RSTART** and **RLENGTH.** **RSTART** is set to the starting position of the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. For example, running the program

```
{ match($0, "ia")
   if (RSTART != 0) print RSTART, RLENGTH
}
```

produces the following output:

```
17 2
18 2
8 2
4 2
6 2
```

The function **sprintf**(*format*, $expr_1$, $expr_2$, ..., $expr_n$) returns (without printing) a string containing $expr_1$, $expr_2$, ..., $expr_n$ formatted according to the **printf** specifications in the string *format*. "The **printf** Statement" in this chapter contains a complete specification of the format conventions. The statement

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to `x` the string produced by formatting the values of `$1` and `$2` as a ten-character string and a decimal number in a field of width at least six; `x` may be used in any subsequent computation.

The function **substr**(*s,p,n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s,p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in **countries** to their first three characters by invoking the program

```
{ $1 = substr($1, 1, 3); print }
```

on this file to produce

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting `$1` in the program forces **awk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file **countries,**

```
                    { s = s substr($1, 1, 3) " " }
END                 { print s }
```

prints

```
USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

by building `s` up a piece at a time from an initially empty string.

Field Variables

The fields of the current record can be referred to by the field variables **$1, $2, ..., $NF.** Field variables share all of the properties of other variables — they may be used in arithmetic or string operations, and may be assigned. So, for example, you can divide the second field of the file **countries** by 1000 to convert the area from thousands to millions of square miles:

```
{ $2 /= 1000; print }
```

or assign a new string to a field:

```
BEGIN                    { FS = OFS = "\t" }
$4 == "North America"    { $4 = "NA" }
$4 == "South America"    { $4 = "SA" }
                         { print }
```

The `BEGIN` action in this program resets the input field separator `FS` and the output field separator `OFS` to a tab. Notice that the `print` in the fourth line of the program prints the value of **$0** after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, **$(NF−1)** is the second last field of the current record. The parentheses are needed: the value of **$NF−1** is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, **$(NF+1),** has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file **countries** creates a fifth field giving the population density:

```
BEGIN           { FS = OFS = "\t" }
                { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or String?

Variables, fields and expressions can have both a numeric value and a string value. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like

```
pop += $3
```

pop and $3 must be treated numerically, so their values will be coerced to numeric type if necessary.

In a string context like

```
print $1 ":" $2
```

$1 and $2 must be strings to be concatenated, so they will be coerced if necessary.

In an assignment *v* = *e* or *v op*= *e*, the type of *v* becomes the type of *e*. In an ambiguous context like

```
$1 == $2
```

the type of the comparison depends on whether the fields are numeric or string, and this can only be determined when the program runs; it may well differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "$1 == $2" will succeed on any pair of the inputs

```
1    1.0    +1    0.1e+1    10E-1    1e2    10e1    001
```

but fail on the inputs

(null)	0
(null)	0.0
`0a`	`0`
`1e50`	`1.0e50`

There are two idioms for coercing an expression of one type to the other:

number ""	concatenate a null string to a *number* to coerce it to type string
string + 0	add zero to a *string* to coerce it to type numeric

Thus, to force a string comparison between two fields, say

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

Control Flow Statements

awk provides **if–else, while, do–while,** and **for** statements, and statement grouping with braces, as in C.

The **if** statement syntax is

if (*expression*) *statement*$_1$ **else** *statement*$_2$

The *expression* acting as the conditional has no restrictions; it can include the relational operators <, <=, >, >=, ==, and !=; the regular expression matching operators ~ and !~; the logical operators ||, &&, and !; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the *expression* is first evaluated. If it is non-zero and non-null, *statement*$_1$ is executed; otherwise *statement*$_2$ is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from "Arithmetic Functions" with an **if** statement results in

```
{               if (maxpop < $3) {
                maxpop = $3
                country = $1
                }
}
END             { print country, maxpop }
```

The **while** statement is exactly that of C:

while (*expression*) *statement*

The *expression* is evaluated; if it is non-zero and non-null the *statement* is executed and the *expression* is tested again. The cycle repeats as long as the *expression* is non-zero. For example, to print all input fields one per line,

```
{               i = 1
                while (i <= NF) {
                print $i
                i++
                }
}
```

The **for** statement is like that of C:

for ($expression_1$; *expression*; $expression_2$) *statement*

It has the same effect as

$expression_1$
while (*expression*) {
statement
$expression_2$
}

so

```
{ for (i = 1; i <= NF; i++)  print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the form

do *statement* **while** (*expression*)

The *statement* is executed repeatedly until the value of the *expression* becomes zero. Because the test takes place after the execution of the *statement*, it is always executed at least once.

The **do** statement in the program

```
{ i = 0;
   do { print $i++ }
      while (i <= NF)
}
```

prints each field in a record in a vertical listing. produces the following output when run with the file **countries**:

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit**
statement causes the program to behave as if the end of the input had occurred; no more input is read, and the **END** action, if any, is executed. Within the **END** action,

exit *expr*

causes the program to return the value of *expr* as its exit status. If there is no *expr*, the exit status is zero.

Arrays

awk provides one-dimensional arrays. Arrays and array elements need not be declared; like variables, they spring into existence by being mentioned. An array subscript may be a number or a string.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array `x`. In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program

```
                { x[NR] = $0 }
END             { ... processing ... }
```

The first action merely records each input line in the array x, indexed by line number; processing is done in the **END** statement.

Array elements may also be named by nonnumeric values, a facility that gives **awk** a capability rather like the associative memory of Snobol tables. For example, the following program accumulates the total population of Asia and Africa into the associative array pop . The END in the program action prints the total population of these two continents.

```
/Asia/      { pop["Asia"] += $3 }
/Africa/    { pop["Africa"] += $3 }
END         { print "Asian population in millions is", pop["Asia"]
              print "African population in millions is", pop["Africa"] }
```

On the file **countries,** this program generates

```
Asian population in millions is 1765
African population in millions is 37
```

In this program if we had used pop[Asia] instead of pop["Asia"] the expression would have used the value of the variable Asia as the subscript, and since the variable is uninitialized, the values would have been accumulated in pop[""] .

Suppose our task is to determine the total area in each continent of the file **countries**. Any expression can be used as a subscript in an array reference. Thus

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array area and in that entry accumulates the value of the second field:

```
BEGIN       { FS = "\t" }
            { area[$4] += $2 }
END         { for (name in area)
            print name, area[name] }
```

Invoked on the file **countries,** this program produces

```
South America 4358
Africa 1888
Asia 13611
Australia 2968
North America 7467
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

> **for** (*i* **in** *array*) *statement*

executes *statement* with the variable *i* set in turn to each value of *i* for which *array[i]* has been defined. The loop is executed once for each defined subscript, in a random order. A program does not run properly if *i* is altered during the loop.

awk does not provide multi-dimensional arrays, so you cannot write **x[i,j]** or **x[i][j]**. You can, however, create your own subscripts by concatenating row and column values with a suitable separator. For example,

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
        arr[i "," j] = ...
```

creates an array whose subscripts have the form `i,j`, such as 1,1 or 1,2. (The comma distinguishes a subscript like 1,12 from one like 11,2.)

You can determine whether a particular subscript *i* occurs in an array *arr* by testing the condition *i* in *arr*, as in

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating `area["Africa"]`, which would happen if we used

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array `area` contains an element with value

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The function

```
split("s1:s2:s3", a, ":")
```

splits the string `s1:s2:s3` into three fields, using the separator `:` and storing `s1` in `a[1]`, `s2` in `a[2]`, and `s3` in `a[3]`. The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element may be deleted with the **delete** statement:

> **delete** *arrayname*[*subscript*]

User-Defined Functions

awk provides user-defined functions. A function is defined as

> **func** *name*(*argument-list*) {
> *statements*
> }

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, to define and test the usual recursive factorial function,

```
func fact(n) {
        if (n <= 1)
        return 1
        else
        return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however; so the function cannot affect their values outside. Within a function, formal parameters are local variables but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables; because arrays are passed by reference, however, the local variables can only be scalars.) The **return** statement is optional, but the returned value is undefined if execution falls off the end of the function.

Comments

Comments may be placed in **awk** programs: they begin with the character # and end at the end of the line, as in

```
print x, y # this is a comment
```

Output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output, so that output from **print** and **printf** can be directed to files and pipes. This section describes the use of these two statements.

The print Statement

The statement

print $expr_1$, $expr_2$, ..., $expr_n$

prints the string value of each expression separated by the output field separator followed by the output record separator. The statement

```
print
```

is an abbreviation for

```
print $0.
```

To print an empty line use

```
print "".
```

Output Separators

The output field separator and record separator are held in the built-in variables **OFS** and **ORS**. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but these values can be changed at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN       { OFS = ":"; ORS = "\n\n" }
            { print $1, $2 }
```

Notice that

```
            { print $1 $2 }
```

prints the first and second fields with no intervening output field separator, because `$1 $2` is a string consisting of the concatenation of the first two fields.

The printf Statement

awk 's **printf** statement is the same as that in C except that the **c** and * format specifiers are not supported. The **printf** statement has the general form

printf *format*, $expr_1$, $expr_2$, ..., $expr_n$

where *format* is a string that contains both information to be printed and specifications on what conversions are to be performed on the expressions in the argument list, as in Figure 4-8. Each specification begins with a %, ends with a letter that determines the conversion, and may include

–	left-justify expression in its field
width	pad field to this width as needed; leading 0 pads with zeros
.prec	maximum string width or digits to right of decimal point

Character	Prints Expression as
d	decimal number
e	[-]d.ddddddE[+-]dd
f	[-]ddd.dddddd
g	e or f conversion, whichever is shorter, with nonsignificant zeros suppressed
o	unsigned octal number
s	string
x	unsigned hexadecimal number
%	print a %; no argument is converted

Figure 4-8: **awk** Conversion Characters

Here are some examples of **printf** statements along with the corresponding output:

```
printf "%d", 99/2                      49
printf "%e", 99/2                      4.950000e+01
printf "%f", 99/2                      49.500000
printf "%6.2f", 99/2                   49.50
printf "%g", 99/2                      49.5
printf "%o", 99                        143
printf "%06o", 99                      000143
printf "%x", 99                        63
printf "|%s|", "January"               |January|
printf "%d", 99/2                      49
printf "|%10s|", "January"             |   January|
printf "|%-10s|", "January"            |January   |
printf "|%.3s|", "January"             |Jan|
printf "|%10.3s|", "January"           |       Jan|
printf "|%-10.3s|", "January"          |Jan       |
printf "%%"                            %
```

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into Files

It is possible to print output into files instead of to the standard output by using the > and >> redirection operators. For example, the following program invoked on the file **countries** prints all lines where the population (third field) is bigger than 100 into a file called `bigpop`, and all other lines into `smallpop`:

```
$3 > 100    { print $1, $3 >"bigpop" }
$3 <= 100   { print $1, $3 >"smallpop" }
```

Notice that the filenames have to be quoted; without quotes, `bigpop` and `smallpop` are merely uninitialized variables. If the output files were not literals, they would also have to be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME ) }
```

This is because the > operator has higher precedence than concatenation; without parentheses, the concatenation of `tmp` and `FILENAME` would not work.

NOTE

Files are opened once in an **awk2** program; If >> is used instead of > to open a file, output is appended to the file rather than overwriting its original contents. However, when the file is subsequently written to, the two operators function exactly the same.

Output into Pipes

It is also possible to direct printing into a pipe with a command on the other end, instead of a file. The statement

```
print | "command-line"
```

causes the output of **print** to be piped into the *command-line*.

Although we have shown them here as literal strings enclosed in quotes, the *command-line* and filenames can come from variables, etc., as well.

Suppose we want to create a list of continent-population pairs, sorted alphabetically by continent. The **awk** program below accumulates in an array pop the population values in the third field for each of the distinct continent names in the fourth field, prints each continent and its population, and pipes this output into the sort command.

```
BEGIN   { FS = "\t" }
        { pop[$4] += $3 }
END     { for (c in pop)
          print c ":" pop[c] | "sort" }
```

Invoked on the file **countries,** this program yields

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named sort), but they are created and opened only once in the entire run. So, in the last example, for all c in pop, only one sort pipe is open.

However, let's say we use **egrep**(1) and the pipe command

```
... | ( "egrep " c " file-name" )
                    #note the need for parens here since
                    # otherwise the | has higher precedence
```

Each iteration of this pipe is associated with a new pipe, because c changes in each iteration. As a result, the program complains that it cannot open egrep *<value of c> file* and stops. The entire pipe name needs to be closed to keep the program from failing. The loop needed is

```
for ( c in pop ) {
  ... | ( "egrep " c " file-name" )
  close ( "egrep " c " file-name" )
  ... }
```

Given this requirement, you program would be clearer if you assigned the pipe each time to a variable:

```
for ( c in pop ) {
  pipe_cmd = "egrep " c " file-name"
  ... | pipe_cmd
  close ( pipe_cmd  )
  ...
  }
```

There is a limit to the number of files that can be open simultaneously. The statement **close(***file***)** closes a file or pipe; *file* is the string used to create it in the first place, as in

```
close("sort") .
```

Input

There are several ways to give input to an **awk** program. The most common way is to name on the command line the file that contains the input your program needs. In addition, you can the **getline** function within a program to read in lines. You can also use command line arguments and pipes to provide input. This section describes each of these methods.

Files and Pipes

The most common way to provide input to an **awk** program is to put the data into a file, say *awkdata*, and then execute

awk *'program' awkdata*

awk reads its standard input if no filenames are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **awk** . For example, **egrep**(1) selects input lines containing a specified regular expression, but it can do so faster than **awk** since this is the only thing it does. We could, therefore, invoke the pipe

egrep 'Asia' countries | **awk '...'**

egrep quickly finds the lines containing **Asia** and pass them on to the **awk** program for subsequent processing.

Input Separators

With the default setting of the field separator **FS,** input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
    field1                field2
  field1
field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. For example,

```
BEGIN { FS = "(,[ \\t]*)|([ \\t]+)" } ...'
```

sets it to an optional comma followed by any number of blanks and tabs. **FS** can

also be set on the command line with the **–F** argument:

awk –F'(,[\t]*)|([\t]+)' '...'

behaves the same as the previous example. Regular expressions used as field separators do not match null strings.

Multi-Line Records

Records are normally separated by newlines, so that each line is a record, but this too can be changed, though only in a limited way. If the built-in record separator variable **RS** is set to the empty string, as in

```
BEGIN    { RS = "" }
```

then input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to use

```
BEGIN    { RS = ""; FS = "\n" }
```

to set the record separator to an empty line and the field separator to a newline. There is a limit, however, on how long a record can be; it is usually about 2500 characters. "The **getline** Function" and "Cooperation with the Shell" in this chapter show other examples of processing multi-line records.

The getline Function

awk's limited facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting **RS** to null doesn't work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

The function **getline** can be used to read input either from the current input or from a file or pipe, by redirection analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF, NR,** and **FNR. getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and –1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with `START` and ends with a line beginning with `STOP` . The following **awk** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array

```
f[1] f[2] ... f[nf]
```

Once the line containing STOP is encountered, the record can be processed from the data in the f array:

```
/^START/ {
            f[nf=1] = $0
            while (getline && $0 !~ /^STOP/)
            f[++nf] = $0
            # now process the data in f[1]...f[nf]
            ...
}
```

Notice that this code uses the fact that && evaluates its operands left to right and stops as soon as one is true.

The same job can also be done by the following program:

```
/^START/ && nf==0       { f[nf=1] = $0 }
nf > 1                  { f[++nf] = $0 }
/^STOP/                 { # now process the data in f[1]...f[nf]
                            ...
                            nf = 0
}
```

The statement

```
getline x
```

reads the next record into the variable x. No splitting is done; **NF** is not set. The statement

```
getline <"file"
```

reads from file instead of the current input. It has no effect on **NR** or **FNR**, but field splitting is performed and **NF** is set. The statement

```
getline x <"file"
```

gets the next record from file into x ; no splitting is done, and **NF**, **NR** and **FNR** are untouched.

NOTE

If you use the statement form

```
getline x < "file"
```

to construct a file name of more than one literal or variable, the file name needs to be placed in parentheses for correct evaluation:

```
while ( getline x < ( ARGV[1].ARGV[2] ) ) {  ... }
```

Without parentheses, a statement such as

```
getline x < "tmp".FILENAME
```

sets x to read the file tmp and not tmp.<*value of FILENAME*>. e statement

```
while ( getline x < "tmp" "file" ) { ... }
```

for instance, loops infinitely because x is always set to null.

It is also possible to pipe the output of another command directly into **getline**. For example, the statement

```
while ("who" | getline)
      n++
```

executes who and pipes its output into getline . Each iteration of the while loop reads one more line and increments the variable n, so after the while loop terminates, n contains a count of the number of users. Similarly, the statement

```
"date" | getline d
```

pipes the output of date into the variable d, thus setting d to the current date.

Figure 4-9 summarizes the **getline** function.

Form	Sets
getline	**$0, NF, NR, FNR**
getline *var*	*var*, **NR, FNR**
getline <*file*	**$0, NF**
getline *var* <*file*	*var*
cmd | **getline**	**$0, NF**
cmd | **getline** *var*	*var*

Figure 4-9: **getline** Function

Command-line Arguments

The command-line arguments are available to an **awk** program: the array **ARGV** contains the elements **ARGV[0]**, ..., **ARGV[ARGC-1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **awk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments). The following command line contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
  for (i = 1; i < ARGC; i++)
    printf "%s ", ARGV[i]
  printf "\n"
  exit
}' $*
```

The arguments may be modified or added to; **ARGC** may be altered. As each input file ends, **awk** treats the next non-null element of **ARGV** (up to the current value of **ARGC-1**) as the name of the next input file.

There is one exception to the rule that an argument is a filename: if it is of the form

var=value

then the variable *var* is set to the value *value* as if by assignment. Such an argument is not treated as a filename. If *value* is a string, no quotes are needed.

Using awk with other Commands

awk gains its greatest power when it is used in conjunction with other programs. Here we describe some of the ways in which **awk** programs cooperate with other commands.

The system Function

The built-in function **system**(*command-line*) executes the command *command-line*, which may well be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the status return of the command executed.

For example, the program

```
$1 == "#include"  { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

calls the command `cat` to print the file named in the second field of every input record whose first field is `#include`, after stripping any <, > or " that might be present.

Cooperation with the Shell

In all the examples thus far, the **awk** program was in a file and fetched from there using the **–f** flag, or it appeared on the command line enclosed in single quotes, as in

awk '{ print $1 }' ...

Since **awk** uses many of the same characters as the shell does, such as $ and ", surrounding the **awk** program with single quotes ensures that the shell will pass the entire program unchanged to the **awk** interpreter.

Now, consider writing a command **addr** that will search a file **addresslist** for name, address, and telephone information. Suppose that **addresslist** contains names and addresses in which a typical entry is a multi-line record such as

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

We want to search the address list by issuing commands like

addr Emlin

That is easily done by a program of the form

```
awk '
BEGIN        { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called **addr** that contains

```
awk '
BEGIN        { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here: the **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The `$1` is outside the quotes, visible to the shell, which therefore replaces it by the pattern `Emlin` when the command **addr Emlin** is invoked. On a UNIX system, **addr** can be made executable by changing its mode with the following command: **chmod +x addr.**

A second way to implement **addr** relies on the fact that the shell substitutes for **$** parameters within double quotes:

```
awk "
BEGIN        { RS = \"\" }
/$1/
" addresslist
```

Here we must protect the quotes defining `RS` with backslashes, so that the shell passes them on to **awk**, uninterpreted by the shell. `$1` is recognized as a parameter, however, so the shell replaces it by the pattern when the command **addr** *pattern* is invoked.

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline:**

```
awk '
BEGIN    { RS = ""
           while (getline < "addresslist")
              if ($0 ~ ARGV[1])
                 print $0
           exit
} '
```

All processing is done in the BEGIN action.

Notice that any regular expression can be passed to **addr**; in particular, it is possible to retrieve by parts of an address or telephone number as well as by name.

Example Applications

awk has been used in surprising ways. We have seen **awk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages such as Pascal or C. In this section, we will present a few more examples to illustrate some additional **awk** programs.

Generating Reports

awk is especially useful for producing reports that summarize and format information. Suppose we wish to produce a report from the file **countries** in which we list the continents alphabetically, and after each continent its countries in decreasing order of population:

```
Africa:
              Sudan          19
              Algeria        18

Asia:
              China         866
              India         637
              USSR          262

Australia:
              Australia      14

North America:
              USA           219
              Canada         24

South America:
              Brazil        116
              Argentina      26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, we create a list of continent-country-population triples, in which each field is separated by a colon. This can be done with the following program **triples,** which uses an array `pop` indexed by subscripts of the form "continent:country" to store the population of a given country. The print statement

in the END section of the program creates the list of continent-country-population triples that are piped to the sort routine.

```
BEGIN   { FS = "\t" }
        { pop[$4 ":" $1] += $3 }
END     { for (cc in pop)
              print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for sort deserve special mention. The -t: argument tells sort to use : as its field separator. The +0 -1 arguments make the first field the primary sort key. In general, $+i$ $-j$ makes fields $i+1$, $i+2$, ..., j the sort key. If $-j$ is omitted, the fields from $i+1$ to the end of the record are used. The +2nr argument makes the third field, numerically decreasing, the secondary sort key (n is for numeric, r for reverse order). Invoked on the file **countries,** this program produces as output

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form we run it through a second **awk** program **format**:

```
BEGIN       { FS = ":" }
{           if ($1 != prev) {
            print "\n" $1 ":"
            prev = $1
            }
            printf "\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The command line

awk –f triples countries | awk –f format

gives us our desired report. As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awk** 's and **sort**'s.

As an exercise, add to the population report subtotals for each continent and a grand total.

Additional Examples

1. *Word frequencies.*
   Our first example illustrates associative arrays for counting. Suppose we want to count the number of times each word appears in the input, where a word is any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order.

```
        { for (w = 1; w <= NF; w++) count[$w]++ }
END     { for (w in count) print count[w], w | "sort -nr" }
```

   The first statement uses the array count to accumulate the number of times each word is used. Once the input has been read, the second for loop pipes the final count along with each word into the sort command.

2. *Accumulation.*
   Suppose we have two files, deposits and withdrawals, of records containing a name field and an amount field. For each name we want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == "deposits"      { balance[$1] += $2 }
FILENAME == "withdrawals"   { balance[$1] -= $2 }
END                         { for (name in balance)
                                   print name, balance[name]
} ' deposits withdrawals
```

   The first statement uses the array balance to accumulate the total amount for each name in the file deposits . The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name will be created by the second statement. The END action prints each name with its net balance.

3. *Random choice.*
   The following function prints (in order) k random elements from the first n elements of the array A . In the program, k is the number of entries that still need to be printed, and n is the number of elements yet to be examined. The decision of whether to print the *i*th element is determined by the test rand() < k/n.

```
func choose(A, k, n) {
        for (i = 1; n > 0; i++)
                if (rand() < k/n--) {
                        print A[i]
                        k--
                }
        }
}
```

4. *Shell facility.*
   The following **awk** program simulates (crudely) the history facility of the UNIX system shell. A line containing only = re-executes the last command executed. A line beginning with = *cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

```
$1 == "=" { if (NF == 1)
                system(x[NR] = x[NR-1])
            else
                for (i = NR-1; i > 0; i--)
                        if (x[i] ~ $2) {
                                system(x[NR] = x[i])
                                break
                        }
            next }

/./       { system(x[NR] = $0) }
```

5. *Form-letter generation.*
   The following program generates form letters, using a template stored in a file called fform.letter :

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

and replacement text of this form:

```
field 1|field 2|field 3
one|two|three
a|b|c
```

The `BEGIN` action stores the template in the array `template` ; the remaining action cycles through the input data, using `gsub` to replace template fields of the form `$f2` with the corresponding data fields.

```
BEGIN {FS = "|"
       while (getline <"form.letter")
              line[++n] = $0
}
{      for (i = 1; i <= n; i++) {
              s = line[i]
              for (j = 1; j <= NF; j++)
                     gsub("\\$"j, $j, s)
              print s
       }
}
```

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk Summary

Command Line

awk *program filenames*
awk **–f** *program-file filenames*
awk **–F***s* set field separator to string *s*; **–Ft** set separator to tab

Patterns

BEGIN
END
/regular expression/
relational expression
pattern **&&** *pattern*
pattern || *pattern*
(*pattern*)
!*pattern*
pattern, *pattern*
func *name*(*parameter list*) { *statement* }

Control Flow Statements

if (*expr*) *statement* [**else** *statement*]
if (*subscript* **in** *array*) *statement* [**else** *statement*]
while (*expr*) *statement*
for (*expr*; *expr*; *expr*) *statement*
for (*var* **in** *array*) *statement*
do *statement* **while** (*expr*)
break
continue
next
exit [*expr*]
function-name(*expr*, *expr*, ...)
return [*expr*]

Input-Output

close(*filename*)	close file
getline	set **$0** from next input record; set **NF, NR, FNR**
getline <*file*	set **$0** from next record of *file*; set **NF**
getline *var*	set *var* from next input record; set **NR, FNR**
getline *var* <*file*	set *var* from next record of *file*
print	print current record
print *expr-list*	print expressions
print *expr-list* >*file*	print expressions on *file*
printf *fmt*, *expr-list*	format and print
printf *fmt*, *expr-list* >*file*	format and print on *file*
system(*cmd-line*)	execute command *cmd-line*, return status

In **print** and **printf** above, >>*file* appends to the *file*, and | *command* writes on a pipe. Similarly, *command* | **getline** pipes into **getline**. **getline** returns 0 on end of file, and −1 on error.

String Functions

gsub(*r*,*s*,*t*)	substitute string *s* for each substring matching regular expression *r* in string *t*, return number of substitutions; if *t* omitted, use **$0**
index(*s*,*t*)	return index of string *t* in string *s*, or 0 if not present
length(*s*)	return length of string *s*
match(*s*, *r*)	return position in *s* where regular expression *r* occurs, or 0 if *r* is not present
split(*s*,*a*,*r*)	split string *s* into array *a* on regular expression *r*, return number of field if *r* omitted, **FS** is used in its place
sprintf(*fmt*, *expr-list*)	print *expr-list* according to *fmt*, return resulting string
sub(*r*,*s*,*t*)	like gsub except only the first matching substring is replaced
substr(*s*,*i*,*n*)	return *n*-char substring of *s* starting at *i*; if *n* omitted, use rest of *s*

Arithmetic Functions

atan2(*y*,*x*)	arctangent of *y* / *x* in radians
cos(*expr*)	cosine (angle in radians)
exp(*expr*)	exponential
int(*expr*)	truncate to integer
log(*expr*)	natural logarithm
rand()	random number between 0 and 1
sin(*expr*)	sine (angle in radians)
sqrt(*expr*)	square root
srand(*expr*)	new seed for random number generator; use time of day if no *expr*

Operators (increasing precedence)

= += -= *= /= %= ^=	assignment
||	logical OR
&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
blank	string concatenation
+ -	add, subtract
* / %	multiply, divide, mod
+ - !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
$	field

Regular expressions (increasing precedence)

c	matches non-metacharacter *c*
c	matches literal character *c*
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[*abc*...]	character class matches any of *abc*...
[^*abc*...]	negated class matches any but *abc*... and newline
r1|*r2*	matches either *r1* or *r2*
r1r2	concatenation: matches *r1*, then *r2*
r+	matches one or more *r*s
*r**	matches zero or more *r*s
r?	matches zero or one *r*s
(*r*)	grouping: matches *r*

Built-in Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0..ARGC−1)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default `%.6g`)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	set by **match()**
RLENGTH	set by **match()**

Limits

Any particular implementation of **awk** enforces some limits. Here are typical values:

100 fields
2500 characters per input record
2500 characters per output record
1024 characters per individual field
1024 characters per printf string
400 characters maximum quoted string
400 characters in character class
15 open files
1 pipe
numbers are limited to what can be represented on the local machine, e.g., 1e–38..1e+38

Initialization, Comparison, and Type Coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the assignment

var = *expr*

its type is set to that of the expression. (Assignment includes +=, –=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in

```
v1 = v2
```

then the type of `v1` becomes that of `v2` .

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges, such as

```
expr + 0
```

and to string by

```
expr ""
```

(i.e., concatenation with a null string).

Uninitialized variables have the numeric value **0** and the string value "" . Accordingly, if x is uninitialized,

```
if (x) ...
```

is false, and

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

are all true. But note that

```
if (x == "0") ...
```

is false.

The type of a field is determined by context when possible; for example,

```
$1++
```

clearly implies that $1 is to be numeric, and

```
$1 = $1 "," $2
```

implies that $1 and $2 are both to be strings. Coercion is done as needed.

In contexts where types cannot be reliably determined, for example,

```
if ($1 == $2) ...
```

the type of each field is determined on input. All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value "" ; they are not numeric. Non-existent fields (i.e., fields past **NF**) are treated this way, too.

As it is for fields, so it is for array elements created by **split()**.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if arr[i] does not currently exist,

```
if (arr[i] == "") ...
```

causes it to exist with the value "" so the **if** is satisfied. The special construction

```
if (i in arr) ...
```

determines if arr[i] exists without the side effect of creating it if it does not.

An Overview of lex Programming

lex is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled. The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. In the compiler writing case, creating the ability to solve the problem requires implementing the compiler's lexical analyzer. Hence the name **lex**.

It is not essential to use **lex** to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what **lex** does is produce such C programs. (**lex** is therefore called a program generator.) What **lex** offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In some applications this is a minor consideration, and the advantages of using **lex** considerably outweigh it.

To understand what **lex** does, see the diagram in Figure 5-1. We begin with the **lex** source (often called the **lex** specification) that you, the programmer, write to solve the problem at hand. This **lex** source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the **lex** program generator. The output of the program generator is a C program that, in turn, must be compiled by a host language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic—user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

lex can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see

- how to write **lex** source to do some of these tasks
- how to translate **lex** source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that **lex** provides.

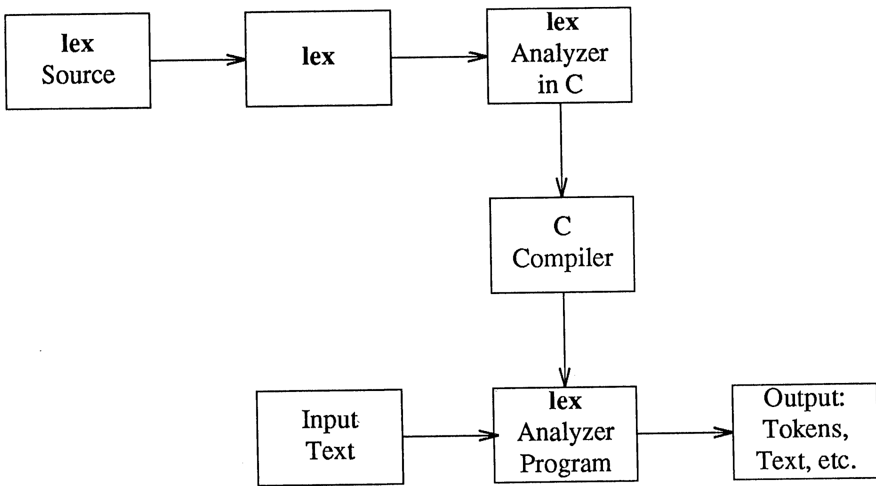


Figure 5-1: Creation and Use of a Lexical Analyzer with **lex**

Writing lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamental lex Rules

The mandatory rules section opens with the delimiter %%. If a subroutines section follows, another %% delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term specification—it may mean either the entire **lex** source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, **lex** writes out the input exactly as it finds it. So, the simplest **lex** program is just the beginning rules delimiter, %%. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in with a notation called regular expressions. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all. For example,

apple
orange
pluto

These three regular expressions match any occurrences of those character strings in an input text. If you want to have your lexical analyzer **a.out** remove every occurrence of **orange**, from the input text, you could specify the rule

```
orange;
```

Because you did not specify an action on the right (before the semi-colon), **lex** does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string **orange** at all.

Unlike **orange** above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite. Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The + operator, for instance, means one or more

occurrences of the preceding expression, the **?** means 0 or 1 occurrence(s) of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and means 0 or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. We will see an example in a moment.) So **m+** is a regular expression matching any string of **m**s such as each of the following:

```
mmm
m
mmmmm
mm
```

and **7*** is a regular expression matching any string of zero or more 7s:

```
77
77777

777
```

The string of blanks on the third line matches simply because it has no 7s in it at all.

Brackets, [ ], indicate any one character from the string of characters specified between the brackets. Thus, **[dgka]** matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma here would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen, –. The sequence **[a-z]**, for instance, indicates any lowercase letter. Somewhat more interestingly,

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether uppercase or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a *, would match any digit or letter. The two pairs of brackets with their enclosed characters

would then match any letter followed by a digit or a letter. But with the asterisk, *, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

```
e
pay
distance
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers:

```
not_idenTIFER
5times
$hello
```

because **not_idenTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them as characters to look for in our search pattern. The last example, for instance, will not recognize text with an * in it. **lex** solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a \ is taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, an * followed by any number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a \ itself, we need two backslashes: \\

Actions

Once **lex** recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression Amelia Earhart and to note such recognition, the rule

```
"Amelia Earhart"   printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. **lex** uses the standard escape sequences from C like **\n** for end-of-line. To count lines we might have

```
\n   lineno++;
```

where **lineno**, like other C variables, is declared in the definitions section that we discuss later.

lex stores every character string that it recognizes in a character array called **yytext[ ]**. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you choose to) write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum, here) and print out each one as soon as it is found, your **lex** code might be

```
+?[1-9]+          { digstrngcount++;
                    printf("%d",digstrngcount);
                    printf("%s", yytext);   }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the **?** indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign, –, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that lets you process input text riddled with quite complicated patterns. These include rules that decide what specification is relevant, when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining an example drawing together several of the points already covered.

```
%%
-[0-9]+          printf("negative integer");
+?[0-9]+         printf("positive integer");
-0.[0-9]+        printf("negative fraction, no whole number part");
rail[ ]+road     printf("railroad is one word");
crook            printf("Here's a crook");
function         subprogcount++;
G[a-zA-Z]*      { printf("may have a G word here: ", yytext);
                  Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and −1. The use of the terminating + in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for **railroad** matches cases where one or more blanks intervene between the two syllables of the word. In the cases of **railroad** and **crook**, you may have simply printed a synonym rather than the messages stated. The rule recognizing a **function** simply increments a counter. The last rule illustrates several points:

- The braces specify an action sequence extending over several lines.
- Its action uses the **lex** array **yytext**[], which stores the recognized character string.
- Its specification uses the * to indicate that zero or more letters may follow the **G**.

Some Special Features

Besides storing the recognized character string in **yytext**[], **lex** automatically counts the number of characters in a match and stores it in the variable **yyleng**. You may use this variable to refer to any specific character just placed in the array **yytext**[]. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) in a just recognized integer, you might write

```
[1-9]+          {if  (yyleng > 2)
                 printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. *Prima facie*, any reserved word, for instance, could match two rules. In the lexical analyzer example developed later in the section on **lex** and **yacc**, the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

NOTE

lex follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize > and >= . If the text has the string >= at one point, you might worry that the lexical analyzer would stop as soon as it recognized the > character to execute the rule for > rather than read the next character and execute the rule for >=.

NOTE

lex follows the rule that it matches the longest character string possible and executes the rule for that.

Here it would recognize the >= and act accordingly. As a further example, the rule would enable you to distinguish + from ++ in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you've in fact found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the DO statement in FORTRAN. In the statement

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index **k** until we read the first comma. Until then, we might have the assignment statement

```
DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the forward-looking slash, / (not the backslash, \), which signifies that what follows is trailing context, something not to be stored in **yytext**[], because it is not part of the token itself. So the rule to recognize the FORTRAN DO statement could be

```
30/[ ]*[0-9][ ]*[a-z A-Z0-9]+=[a-z A-Z0-9]+,    printf("found
DO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

lex uses the **$** as an operator to mark a special trailing context—the end of line. (It is therefore equivalent to **\n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

```
[ \t]+$     ;
```

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, ^, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ]      printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks— **input()**, **unput(c)**, and **output(c)**, respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input()**, thus:

```
\"        while (input() != '"');
```

Upon finding the first double quotation mark, the generated **a.out** will simply continue reading all subsequent characters so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you may use standard I/O routines in C to rewrite the functions **input()**, **unput(c)**, and **output**. These and other programmer-defined functions should be placed in your subroutine section. Your new routines will then replace the standard ones. The standard **input()**, in fact, is equivalent to **getchar()**, and the standard **output(c)** is equivalent to **putchar(c)**.

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include **yymore()**, **yyless(n)**, and **REJECT**. Recall that the text matching a given specification is stored in the array **yytext[]**. In general, once the action is performed for the specification, the characters in **yytext[]** are overwritten with succeeding characters in the input stream to form the next match. The function **yymore()**, by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext[]**. This lets you do one thing and then another, when one string of characters is significant and a longer one including the first is significant as well. Consider a character string bound by **B**s and interspersed with one at an arbitrary location.

```
B...B...B
```

In a simple code deciphering situation, you may want to count the number of characters between the first and second **B**s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is

```
B[^B]*          { if (flag = 0)
                     save = yyleng;
                     flag = 1;
                     yymore();
                else      {
                     importantno = save + yyleng;
                     flag = 0; }
                }
```

where **flag, save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyless(*n*)** lets you reset the end point of the string to be considered to the *n*th character in the original **yytext**[]. Suppose you are again in the code deciphering business and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say uppercase or lowercase **Z**. The code you want might be

```
[a-yA-Y]+[Zz]        {  yyless(yyleng/2);
                     ... process first half of string... }
```

Finally, the function REJECT lets you more easily process strings of characters even when they overlap or contain one another as parts. REJECT does this by immediately jumping to the next rule and its specification without changing the contents of **yytext**[ ]. If you want to count the number of occurrences both of the regular expression **snapdragon** and of its subexpression **dragon** in an input text, the following will do:

```
snapdragon      {countflowers++; REJECT;}
dragon          countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions **comedian** and **diana**, even where the input text has sequences such as **comediana..**:

```
comedian       {comiccount++; REJECT;}
diana          princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, the counters and other necessary variables are declared in the definitions section commencing the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later.

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it will be accessible in the routine, say a parser, that calls it. If, on the other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself right after the left brace, { .

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file to be included in every file that needs them. One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#defines** for token names. Like the declarations, **#include** statements should come between %{ and }%, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the %} that ends your **#include**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within braces.

NOTE

The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:

```
D              [0-9]
L              [a-zA-Z]
B              [ ]
%%
-{D}+           printf("negative integer");
+?{D}+          printf("positive integer");
-0.{D}+         printf("negative fraction");
G{L}*           printf("may have a G word here");
rail{B}+road     printf("railroad is one word");
crook           printf("criminal");
 \"\./{B}+      printf(".\"");
   .               .
   .               .
```

Subroutines

You may want to use subroutines in **lex** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function **put_in_tabl()**, to be discussed in the next section on **lex** and **yacc**, is a good candidate for a subroutine.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between /* and */ :

```
"/*"                    skipcmnts();
.
.               /* rest of rules */
%%
skipcmnts()
{
        for(;;)
        {
           while (input() != '*');
           if (input() != '/') {
                 unput(yytext[yyleng-1]);
           else return;
        }
 }
```

There are three points of interest in this example. First, the **unput(c)** function (putting back the last character read) is necessary to avoid missing the final / if the comment ends unusually with a **/ . In this case, eventually having read an *, the analyzer finds that the next character is not the terminal / and must read some more. Second, the expression **yytext[yyleng–1]** picks out that last character read. Third, this routine assumes that the comments are not nested. (This is indeed the case with the C language.) If, unlike C, they are nested in the source text, after **input**()ing the first */ ending the inner group of comments, the **a.out** will read the rest of the comments as if they were part of the input to be searched for patterns.

Other examples of subroutines would be programmer-defined versions of the I/O routines **input**(), **unput(c)**, and **output**(), discussed above. Subroutines such as these that may be exploited by many different programs would probably do best to be stored in their own individual file or library to be called as needed. The appropriate **#include** statements would then be necessary in the definitions section.

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read this section because it covers information of interest to all **lex** programmers.

The lexical analyzer that **lex** generates (not the file that stores it) takes the name **yylex()**. This name is convenient because **yacc** calls its lexical analyzer by this very name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want to end each **lex** action with the statement **return** *token*, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called **y.tab.c** by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >. Consider the following portion of **lex** source for a lexical analyzer for some programming language perhaps slightly reminiscent of Ada:

```
begin                   return(BEGIN);
end                        return(END);
while                      return(WHILE);
if                         return(IF);
package                    return(PACKAGE);
reverse                    return(REVERSE);
loop                       return(LOOP);
[a-zA-Z][a-zA-Z0-9]*     { tokval = put_in_tabl();
                             return(IDENTIFIER); }
[0-9]+                   { tokval = put_in_tabl();
                             return(INTEGER); }
\+                      { tokval = PLUS;
                             return(ARITHOP); }
\-                     { tokval = MINUS;
                             return(ARITHOP); }
>                        { tokval = GREATER;
                              return(RELOP); }
>=                       { tokval = GREATEREQL;
                             return(RELOP); }
```

Despite appearances, the tokens returned, and the values assigned to **tokval**, are indeed integers. Good programming style dictates that we use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C. For example,

```
 #define BEGIN  1
#define END  2
   .
#define PLUS 7
   .
```

If the need arises to change the integer for some token type, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to insert the statement

```
#include y.tab.h
```

into the definitions section of your **lex** source. The file **y.tab.h** provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, the user written routime **put_in_tabl()** places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl()** assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl()** would be a routine that the compiler writer might place in the subroutines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable PLUS, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a + sign is recognized, the action assigns to **tokval** the value 7, which indicates the +. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by ARITHOP or RELOP).

Running lex Under the UNIX System

As you review the following few steps, you might recall Figure 5-1 at the start of the chapter. To produce the lexical analyzer in C, run

lex lex.l

where **lex.l** is the file containing your **lex** specification. The name **lex.l** is conventionally the favorite, but you may use whatever name you want. The output file that **lex** produces is automatically called **lex.yy.c**; this is the lexical analyzer program that you created with `lex`. You then compile and link this as you would any C program, making sure that you invoke the **lex** library with the **–ll** option:

cc lex.yy.c –ll

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you need not supply your own **main()**.

If you have the **lex** specification spread across several files, you can run **lex** with each of them individually, but be sure to rename or move each **lex.yy.c** file (with **mv**) before you run **lex** on the next one. Otherwise, each will overwrite the previous one. Once you have all the generated **.c** files, you can compile all of them, of course, in one command line.

With the executable **a.out** produced, you are ready to analyze any desired input text. Suppose that the text is stored under the filename **textin** (this name is also arbitrary). The lexical analyzer **a.out** by default takes input from your terminal. To have it take the file **textin** as input, simply use redirection, thus:

a.out < textin

By default, output will appear on your terminal, but you can redirect this as well:

a.out < textin > textout

In running **lex** with **yacc**, either may be run first.

yacc –d grammar.y
lex lex.l

spawns a parser in the file **y.tab.c**. (The **–d** option creates the file **y.tab.h**, which contains the **#define** statements that associate the **yacc** assigned integer token values with the user-defined token names.) To compile and link the output files produced, run

cc lex.yy.c y.tab.c –ly –ll

Note that the **yacc** library is loaded (with the **–ly** option) before the **lex** library (with the **–ll** option) to ensure that the **main()** program supplied will call the **yacc** parser.

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument. If you care to see the C program, **lex.yy.c**, that **lex** generates on your terminal (the default output device), use the **–t** option.

lex –t lex.l

The **–v** option prints out for you a small set of statistics describing the so-called finite automata that **lex** produces with the C program **lex.yy.c**. (For a detailed account of finite automata and their importance for **lex**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 500. If your **lex** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing another entry in the definitions section of your **lex** source, as follows:

```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. (The **–v** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *IRIS-4D Programmer's Reference Manual* page on **lex** for a list of all the options available with the **lex** command. In addition, review the paper by Lesk (the originator of **lex**) and Schmidt, "Lex—A Lexical Analyzer Generator," in volume 5 of the *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1986. It is somewhat dated, but offers several interesting examples.

This tutorial has introduced you to **lex** programming. As with any programming language, the way to master it is to write programs and then write some more.

An Overview of yacc Programming

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes:

- a set of rules to describe the elements of the input
- code to be invoked when a rule is recognized
- either a definition or declaration of a low-level routine to examine the input

yacc then turns the specification into a C language function that examines the input stream. This function, called a parser, works by calling the low-level input scanner. The low-level input scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule, (an action) is invoked. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be

```
date  :  month_name  day  ','  year   ;
```

where **date**, **month_name**, **day**, and **year** represent constructs of interest; presumably, **month_name**, **day**, and **year** are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means 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 evaluating the input. With proper definitions, the input

```
July  4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs 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. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. 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 often can 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 constructs that 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 remainder of this chapter describes the following subjects:

- basic process of preparing a **yacc** specification
- parser operation
- handling ambiguities
- handling operator precedences in arithmetic expressions
- error detection and recovery
- the operating environment and special features of the parsers **yacc** produces
- suggestions to improve the style and efficiency of the specifications
- advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs, *% %* (the percent sign is generally used in **yacc** specifications as an escape character).

A full specification file looks like:

declarations
%%
rules
%%
subroutines

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest legal **yacc** specification is

%%
rules

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A  :  BODY  ;
```

where A represents a nonterminal symbol, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length and may be made up of letters, dots, underscores, and digits although a digit may not be the first character of a name. Uppercase and lowercase 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 the C language, the backslash, \, is an escape character within literals, and all the C language escapes are recognized. Thus:

'\n'	newline
'\r'	return
'\''	single quote (')
'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'*xxx*'	*xxx* in octal notation

are understood by **yacc**. For a number of technical reasons, the NULL 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 is 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
    ;
```

by using the vertical bar. 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 more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
epsilon :    ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon.**

Names representing tokens must be declared. This is most simply done by writing

```
%token   name1  name2 ...
```

in the declarations section. 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, the start symbol has particular importance. 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 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 end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end of file or end of record.

Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. 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 language statement and as such can do input and output, call subroutines, and alter arrays 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
         {
            (void) printf("a message\n");
            flag = 25;
         }
```

are grammar rules with actions.

The dollar sign symbol, $, is used to facilitate communication between the actions and the parser, The pseudo-variable $$ represents the value returned by the complete action. For example, the action

```
{  $$ = 1;  }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables **$1, $2, ...** $*n*. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is

```
A  :  B  C  D  ;
```

then **$2** has the value returned by **C**, and **$3** the value returned by **D**.

The rule

```
expr   :   '('  expr  ')'   ;
```

provides a common example. One would expect the value returned by this rule to be the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result 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 previous examples, all the actions came at the end of 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 action 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 below the effect is to set **x** to 1 and **y** to the value returned by **C**.

```
A   :   B
            {
                $$ = 1;
            }
            C
    {
            x = $2;
            y = $3;
    }
    ;
```

Actions that do not terminate a rule are 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 by recognizing this added rule. **yacc** treats the above example as if it had been written

```
$ACT    :   /* empty */
        {
            $$ = 1;
        }
        ;

A       :   B  $ACT  C
        {
            x = $2;
            y = $3;
        }
        ;
```

where **$ACT** is an empty action.

In many applications, output is not done directly by the actions. 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 a 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 can be made known to the lexical analyzer. For example:

```
%{   int variable = 0;    %}
```

could be placed in the declarations section making **variable** accessible to all of the actions. Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far all the values are integers. A discussion of values of other types is found in the section "Advanced **yacc** Features." This section discusses advanced uses of **yacc**.

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 the user. In either case, the **#define** mechanism of C language 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

```
int yylex()
{
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch (c)
        {
           ...
           case '0':
           case '1':
           ...
           case '9':
           yylval = c - '0';
           return (DIGIT);
           ...
        }
        ...
}
```

to return the appropriate token.

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 subroutines section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language 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.

In the default situation, token 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. If the **yacc** command is invoked with the **–d** option a file called **y.tab.h** is generated. **y.tab.h** contains **#define** statements for the tokens.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a non-negative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker 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 a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** 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.

Parser Operation

yacc turns the specification file into a C language procedure, 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. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

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 look-ahead 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 look-ahead token has been read.

The machine has only four actions available—**shift**, **reduce**, **accept**, and **error**. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead 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 look-ahead 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 look-ahead token. For example, in state 56 there may be an action

```
IF   shift 34
```

which says, in state 56, if the look-ahead 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 look-ahead 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 look-ahead token to decide whether or not to **reduce** (usually it is not necessary). In fact, the default action (represented by a dot) is often a **reduce** action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads 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

```
A  :  x y z   ;
```

is being reduced. 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 look-ahead token is cleared by a shift but 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 stacks. 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 look-ahead token is the end-marker 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 look-ahead 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 discussed later.

Consider:

```
%token  DING  DONG  DELL
%%
rhyme   :   sound  place
        ;
sound   :   DING  DONG
        ;
place   :   DELL
        ;
```

as a **yacc** specification.

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) follows.

```
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
```

The actions for each state are specified and 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. The following input

```
DING  DONG  DELL
```

can be used to track the operations of the parser. 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 and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by

```
sound  :  DING  DONG
```

which is rule 2. 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. State 2 is pushed onto the stack and becomes 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 look-ahead 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 and the end-marker is obtained indicated by **$end** in the **y.output** file. The action in state 1 (when the end-marker is seen) successfully ends 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 spent with this and other simple examples is repaid when problems arise in more complicated contexts.

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. Given the input

```
expr - expr - expr
```

consider the problem that confronts the parser. When the parser has read the second *expr*, the input 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, if the parser sees

```
expr - expr
```

it could defer the immediate application of the rule and continue reading the input until

```
expr - expr - expr
```

is seen. It could then apply the rule to the rightmost three symbols reducing them to *expr*, which results in

```
expr  -  expr
```

being left. 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 one of two legal things, a shift or a reduction. It 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 the choice to make in a given situation is called a disambiguating rule.

yacc invokes two default disambiguating rules:

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 **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when 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

```
stat   :  IF  '('  cond  ')'  stat
       |  IF  '('  cond  ')'  stat  ELSE  stat
       ;
```

which is a fragment from a programming language involving an **if-then-else** statement. 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 because 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
}
```

where 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, such as

```
IF  ( C1 )  IF  ( C2 )  S1
```

have already been seen. 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
```

where the first line describes the conflict—giving the state and the input symbol. The ordinary state description gives 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
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot, .), is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not ELSE, the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

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 in parentheses after those rules, which can be reduced. In most states, there is a reduce action possible in the state and this is 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.

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) ) )
```

in order to perform the correct precedence of operators. 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 minus, –.

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. The keyword **%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, the rules

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr   :   expr  '+'  expr
       |   expr  '-'  expr
       |   expr  '*'  expr
       |   expr  '/'  expr
       |   '-'  expr      %prec  '*'
       |   NAME
       ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. 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 default 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 precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; 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 a 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.

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/or, 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 the token name **error**. 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 look-ahead token and performs the action encountered. The look-ahead 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
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans 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. Rules such as

```
stat    :    error ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon 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. The following example

```
input   :   error  '\n'
                {
                    (void) printf( "Reenter last line: " );
                }
                input
            {
                $$ = $4;
            }
            ;
```

is one way to do this. 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 force the parser to believe that error recovery has been accomplished. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input   :   error  '\n'
                {
                   yyerrok;
                   (void) printf( "Reenter last line: " );
                }
                input
            {
               $$ = $4;
            }
            ;
```

which is somewhat better.

As previously mentioned, 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 look-ahead 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 is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset. A rule similar to

```
stat    :   error
        {
           resynch();
           yyerrok  ;
           yyclearin;
        }
        ;
```

could perform this.

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.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language subroutines, called **y.tab.c**. 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 "Lexical Analysis"), to obtain input tokens. Eventually, an error is detected, **yyparse**() returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker 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 language program, a routine called **main**() must be defined that eventually calls **yparse**(). In addition, a routine called **yyerror**() is needed to print 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 **yerror**(). The library is accessed by a **–ly** argument to the **cc**(1) command or to the loader. The source codes

```
main()
{
     return (yyparse());
}
```

and

```
# include <stdio.h>

yyerror(s)
        char *s;
{
        (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs. The argument to **yerror**() is a string containing an error message, usually the string **syntax error**. The average application wants 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 look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main**() routine 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 the input symbols read and what the parser actions are. It is possible to set this variable by using the debugger **edge**.

Hints for Preparing Specifications

This part 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 are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for non-terminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. 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.
4. 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.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 is written following this style, as are the examples in this section (where space permits). The user must decide 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  ;
```

match this algorithm. These rules such as

```
list    :    item
        |    list  ','  item
        ;
```

and

```
seq     :   item
        |   seq  item
        ;
```

frequently arise when writing specifications of sequences and lists. 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 is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq     :   /* empty */
        |   seq  item
        ;
```

using an empty rule. 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 these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
   int dflag;
%}
  ... other declarations ...

%%

prog    :   decls  stats
        ;

decls   :   /* empty */
        {
               dflag = 1;
        }
        |   decls  declaration
        ;

stats   :   /* empty */
        {
               dflag = 0;
        }
        |   stats  statement
        ;

    ... other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. 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 back-door 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 you 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, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced yacc Features

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. The YYACCEPT macro 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 end-markers 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.

```
sent    :   adj  noun  verb  adj  noun
        {
            look at the sentence ...
        }
        ;
adj     :   THE
        {
                $$ = THE;
        }
        |   YOUNG
        {
                $$ = YOUNG;
        }
        ...
        ;
noun    :   DOG
        {
            $$ = DOG;
        }
        |   CRONE
        {
            if( $0 == YOUNG )
            {
                (void) printf( "what?\n" );
            }
            $$ = CRONE;
        }
        ;
        ...
```

In this case, the digit may be 0 or negative. 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 prevents 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 is strictly type checked. **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with 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 take place. In addition, type checking commands such as **lint** are 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 subroutines, 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 cannot easily determine the type.

To declare the union, the user includes

```
%union
{
   body of union ...
}
```

in the declaration section. 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 as YYSTYPE.

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>  '+'  '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **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**) 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 $. The example

```
rule    :   aaa
                {
                   $<intval>$ = 3;
                }
                bbb
        {
            fun( $<intval>2, $<other>0 );
        }
        ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. 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 **ints**.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies, etc. are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, 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 decides whether the next token (skipping

blanks, newlines, and 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 entries */
%token     IDENTIFIER    /* includes identifiers and literals */
%token     C_IDENTIFIER  /* identifier (but not literal) followed by a : */
%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 */

%token     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
        |   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 */
        ;

   /* rule 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 ';'
        ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the operators

```
+, -, *, /, %  (mod operator), & (bitwise and), | (bitwise or),
and assignments.
```

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, 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 precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer 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 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
          {
            (void) 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
          {
              $$ = $1 / $3;
          }
          |  exp '%' expr
          {
               $$ = $1 % $3;
          }
          |  expr '&' expr
          {
              $$ = $1 & $3;
          }
          |  expr '|' expr
          {
```

```
                $$ = $1 | $3;
          }
          |  '-' expr  %prec UMINUS
          {
                $$ = -$2;
          }
          |  LETTER
          {
                $$ = reg[$1];
          }
          |  number
          ;

number    :  DIGIT
          {
                $$ = $1; base = ($1==0) ? 8 ; 10;
          }
          |  number DIGIT
          {
                $$ = base * $1 + $2;
          }
          ;

%%                 /* beginning of subroutines section */

int yylex( )    /* lexical analysis routine */
{               /* return LETTER for lowercase letter, */
                /* yylval = 0 through 25 */
                /* returns DIGIT for digit, yylval = 0 through 9 */
                /* all other characters are returned immediately */
```

```
        int c;
                /*skip blanks*/
        while ((c = getchar()) == ' ')
                ;

                /* c is now nonblank */

        if (islower(c))
        {
                yylval = c - 'a';
                return (LETTER);
        }
        if (isdigit(c))
        }
                yylval = c - '0';
                return (DIGIT);

        }
        return (c);
}
```

2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example in Example 1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations +, – *, /, unary – **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 Example 1; 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 doubles. This structure is given a type name, INTERVAL, by using **typedef**. **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy

depends strongly on being able to assign structures and unions in C language. 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. 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 (for example, scalar or interval) of intermediate expressions. Note that 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 value expression in the second example, but this fact is not known until the comma 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 conflict 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. 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 language 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 line

%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 */

%%                /* beginning of rules section */

lines    :  /* empty */
         |  lines line
         ;
line     :  dexp '\n'
         {
                   (void) printf("%15.8f\n",$1);
         }
         |  vexp '\n'
         {
                   (void) 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 )
```

```
         {
                      (void) 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;
              $$.lo = $1.lo – $3.hi;
        }
        |  dvep '–' vdep
        {
              $$.hi = $1 – $3.lo;
              $$.lo = $1 – $3.hi
        }
        |  vexp '*' vexp
        {
              $$ = vmul( $1.lo,$.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.lo, $1.hi, $3 )
        }
        | '-' vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;$$.lo = -$2.hi
        }
        ! '(' vexp ')'
        }
                $$ = $2
        }
        ;

%%                          /* beginning of subroutines section */

# define BSZ 50   /* buffer size for floating point number */

         /* lexical analysis */

int yylex( )
{
        register int c;

                                  /* skip over blanks */
        while ((c = getchar()) == ' ')
                  ;
        if (isupper(c))
        {
            yylval.ival = c - 'A'
            return (VREG);
        }
        if (islower(c))
```

```
{
    yylval.ival = c - 'a',
    return( DREG );
}

    /* gobble up digits. points, exponents */

if (isdigit(c) || c == '.')
{
    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 = ' ';
         if (cp - buf >= BSZ)
            (void) 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 *,/ routine */
        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.)
        {
             (void) 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));
}
```