REFERENCE MANUAL

COMPILER PACKAGE

Butler W. Lampson

University of California, Berkeley

Document No. 30.60.70

Issued March 18, 1966

Revised May 18, 1966

Contract SD-185

Office of Secretary of Defense

Advanced Research Projects Agency

Washington 25, D.C.

TABLE OF CONTENTS

1.0 Introduction

The compiler package (CP) is a collection of useful POPs, subroutines and conventions which provide a convenient framework for constructing compilers for a wide class of languages. A subroutine is provided to read a line from any input medium, with the facilities of the QED line edit if the input device happens to be the teletype. A second routine converts the source line into an internal representation in which each significant constituent of the source line has been replaced by an integer. POPs are available to make recursive calls on recognizers which attempt to analyze the line. Finally, code can be put onto a list of compiled instructions; insertions are possible at any point on the list, and when the statement is completely analyzed the compiled code can be transcribed into core and, if desired, printed out in symbolic form. A collection of miscellaneous routines provide for error correction, control of panics, initialization, pagination and a limited amount of control over three word/cell forward-chained lists which are used by the code generator.

CP also provides the necessary machinery for preserving the source language of the user's program and for interactively altering both source and compiled code. The basic command language for this purpose is identical to that of QED. The user may extend the list of commands at his discretion.

CP includes the following sections:

1) The standard macro package. This is a large collection of macros which are used in the rest of the package. These macros are not needed in the user's program, although any user may of course take them over if he finds them convenient.

2) The user macro package. These macros provide convenient ways of calling for recognizers and code generation. Symbol and data

definition macros are also included. Most users will want to transfer this package into their program.

3) Syntax recognizer POPs. These provide for recursive calling of subroutines which are usually thought of as recognizers for elements of the syntax of the language under consideration. There are two, one skipping on successful recognition, the other on unsuccessful. For examining the constituents of the source line two analogous operations are available.

4) Error control routines. There is an error POP, which prints out a message, calls a routine to clear the compiler, and returns to the main loop of the compiler.

5) Code generation. The compile POP inserts a word at a specified point on a program list. It also recognizes the special case in which the address of the word refers to another element of the program. An operation is available for inserting a program list at a specified point in another one.

6) A symbol table lookup routine which works on symbols of arbitrary length, and a routine to insert new symbols into this table.

7) Read line. This routine reads one line (up to a carriage return) from any file. It deals correctly with multiple blanks and end of file and recognizes line feed as a line continuation character. If the file is teletype, it allows the user all the facilities of the QED line edit.

8) Pre-pass. This routine decomposes a source line into an internal form which will hopefully be more tractable for the compiler. It assumes the availability of a string storage area into which it can put identifiers, which it then looks up. It will collect floating

point numbers as strings and deliver them to a user-provided routine. Any character may be treated as illegal, ignored, provide an internal identifier or cause a transfer to a user-provided routine. There is considerable control over the treatment of blanks.

9) Input-output. There is a character output routine which generates a call of the line feed routine when it sees carriage return or line feed. The line feed routine counts the number of physical lines output and generates page spacing and page headings every 55 lines. A message printing routine is also available.

10) Initialization. From parameters supplied by the user this routine assigns space for symbol table and string storage, generates free storage lists for the five word cells used by the symbol table routines, assigns space for the compiler's storage and the user's program, initializes various parameters and puts into the symbol table a list of names and values supplied by the user.

11) Panic control. This routine acts as an overlord, clearing the entire state of the system and setting up a fork for it to run in. It responds to panics out of the fork in appropriate fashions.

12) Storage allocation. The system checks for overflow of symbol table, program or string storage and takes appropriate action to obtain more storage, calling on user-provided routines as necessary.

13) Command recognition. The user defines a command table from which the system will recognize commands preceded by addresses in the QED form. A number of commands for printing and altering the symbolic are built into the system.

The remainder of this manual is devoted to a detailed description of CP. We begin with storage allocation and other parameters and proceed to a consideration of the 13 system components outlined above.

2.0 Storage Allocation

The initialization routine, GIN, uses the following parameters, which must be provided by the user:

NOSYMS should contain the initial size of the symbol table. Half this number of words will be allocated for the fixed size table and this many five word blocks will be created and put on a free storage list. If more space is needed the system will attempt to obtain it automatically.

LSS should contain the size of string storage in words. The routine GC will be called if string storage runs out. In this case, if some strings are no longer in use, a garbage collection may be appropriate.

ILCFS should contain the length of the compiler free storage in words. The system will take the maximum of this number and 300 as the size of this storage area. Three words from this area are used by each call of CPL, and three plus the number of local variables by each call of RST or RSF. If more space is needed, the system will attempt to obtain it automatically.

TOP contains the location from which free storage cells will be allocated down.

BFS contains the location from which the fixed length symbol table and the string storage area should be built up.

LPROG contains the initial length of the program area. This space is used for both compiled code and source language text. If more space is needed, the compiler will attempt to obtain it automatically.

The figure on the next page depicts the arrangement of storage immediately after initialization. The three areas at the top indicated by ** can grow. When such growth becomes necessary, the areas lower down must be moved. It therefore follows that

1) New symbols must not be created in the middle of computation, i.e., after the pre-pass. This is because movement of compiler free storage cannot be tolerated if there is anything in it.

2) The user must provide two storage allocation routines:

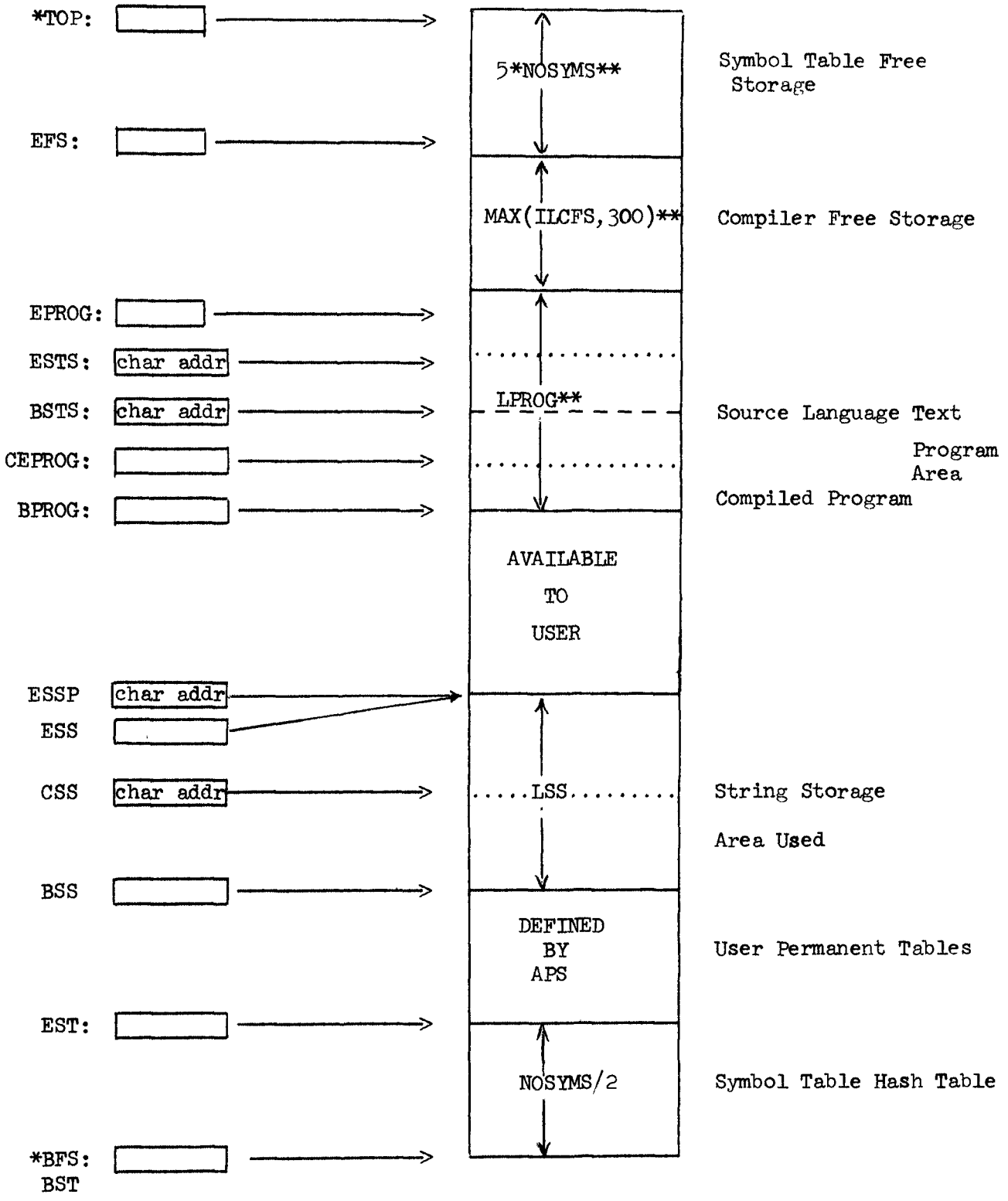
RPROG is called with an address in A and a displacement in B. The routine should make sure that moving the part of the compiled program at or above the specified address by the specified displacement will leave everything in an acceptable state. I.e., it should relocate all addresses which refer to this part of the program.

MSPACE is called with a number in A. The user must ensure that the system can move the beginning of the program, given by BPROG, down by the specified amount.

There are two more user routines connected with storage allocation.

APS is called after CP has assigned its permanent storage (currently the symbol hash table). The user should assign his permanent storage, starting at the location in EST+5, and return the last location he uses in A.

GC should be the string storage garbage collector if the user wishes to do a garbage collection when string storage is exhausted. Otherwise he can put an error routine there.



Storage Allocation for CP. ** indicates expandable area.

After GIN has been called, BST and EST will contain the boundaries of the symbol table, BSS and ESS the word boundaries of string storage, SSP and ESSP the character boundaries of string storage, BPROG and EPROG the boundaries of the program area, RSFS the top of the symbol table free storage list. CSS contains the last used character of string storage.

The following storage allocation and parameterization macros are in the user package:

1) DPS, which is called with a list of opcodes and generates a series of words with labels of the form Zopcode which contain the opcode with 0 address. Thus

```
DPS   ADD,SUB
```

is equivalent to

```
ZADD  ADD   0
ZSUB  SUB   0
```

2) DP is for defining POPs.

```
DP    ADD,55,SUB,54
```

is equivalent to

```
ADD   OPD   5500000B,1
SUB   OPD   5400000B,1
```

3) DSV is for defining symbol values

```
DSV   SYM1,25,SYM2,6400B
```

is equivalent to

```
SYM1  EQU   25
SYM2  EQU   6400B
```

4) DV is for defining values

```
DV    LOC1,100,LOC2,200
```

is equivalent to

```
LOC1   DATA   100
LOC2   DATA   200
```

5) DPOP is for defining POPs whose values have already been set by DSV or EQU. Thus

```
DSV    VPOP1,21,VPOP2,40
DPOP   POP1,POP2
```

is equivalent to

```
POP1   POPD   12100000B,1
POP2   POPD   14000000B,1
```

Note that DPOP expects the symbol containing the POP number to be followed by the POP name.

6) DEF reserves storage

```
DEF    A,B,C
```

is equivalent to

```
$A     ZRO    0
$B     ZRO    0
$C     ZRO    0
```

3.0 Other Parameters

3.1 Listing

If CODFLG is negative, compiled instructions will be listed after they have been put into the program. They are put onto the file indicated by LISTF, which is initialized to 1. The format is one instruction per line, symbolic opcode and numeric address. If either index or indirect bits are on, two digits indicating the state of these two bits will appear after the opcode. CODFLG is in CP temporary storage.

Opcode are taken from two tables called OPTAB for codes 0 to 77 and POPTAB for codes 100 to 177. The appropriate word is picked up and printed as three characters.

Each time RDL is called to read a new line, it will type the character in BLCHAR if the input file is the teletype.

For a description of pagination, see section 7 below.

3.2 Initialization

The initialization routine GIN, in addition to setting up storage as described in the previous section, also sets various flags and counters to their initial values, requests a page heading, and starts off the page numbering with page 1. The flag BRFBEG will suppress the typing associated with initialization if it is negative.

The routine IST, which is called by GIN, initializes the symbol table from the string INS. This string contains a series of entries, each one of which may have either of the following forms:

a) the character /, which indicates that the string is finished;

b) a string of characters which will be taken as a symbol to be looked up, followed by a comma, followed by any number of fields

of the form

<letter> <octal number>

followed by a semi-colon. The octal numbers are collected and interpreted according to the preceding letter as follows:

V set first value word

W set second value word

B set top 8 bits of first name word

C set top 8 bits of second name word

I store index indirect through STIDX + number

For explanation of the terms value word, name word and index, see section 5.1.

STIDX is a table used by I fields as indicated above.

SETUP is a user routine which is called before the processing of each new source line.

To start up CP, call GIN and then transfer to RUBOUT after performing any other initialization required by the particular language being implemented.

3.3 Pre-pass

For a discussion of the meaning of the parameters listed here, see section 8 below:

CHTAB	character table
ICTAB	initial character table
PPSET	pre-pass initialization
PPNL	number lookup
ILCHAR	illegal character exit
IEOS	internal identification of end of statement

3.4 Errors

If the fork control logic is being used

MEMTRP is the location to which control will go after a memory trap;

IITRP is the location to which control will go after an illegal instruction trap.

RERR is called from a few places when CP runs into serious trouble. This is a disaster.

OVFLOW is called when a call of GC returns without skipping.

3.5 The Executive

For a discussion of these parameters, see section 10.

BCT beginning of command table

ECT end of command table

FSTAT find statement

3.6 Miscellaneous

The line edit reads characters from SEIFN and writes them on SEOFN. These locations are built into the system and set to 0 and 1 respectively. A user can, if he wishes, put them in his temporary storage and change them.

TCFIM is the address of a message. Error comments (address of CERRX pops) with addresses smaller than this will be prefixed by this message.

3.7 POP Addresses

At the very beginning of CP is a collection of EQU's which define the values of the various POPs. The user will need this information if he

is to use these POPs in his own program. He may alter the assignments to suit his fancy.

3.8 POP Transfer vector

After a system which includes CP has been loaded, the POP transfer vector must be transcribed into a permanent table, from which it can be rewritten by GIN whenever the system is started. A transfer to FSTART will do this, returning with a BRS 10.

4.0 Syntax Analysis and Code Generation

4.1 Recognizers and Comparers

Syntax analysis in CP is done by recursive calls of routines called recognizers. The design of recognizers is left to the programmer; the system provides only two recursive call POPs:

RST recognize and skip if true

The POP calls the routine addressed. It expects to find at the beginning of this routine a list of addresses terminated with a -1 word. The contents of each word addressed by this list is saved on a linear stack called RPL, together with the address of the call, the address of the recognizer, and the input pointer LBLOC. Control then goes to the word after the -1.

A recognizer returns with a true exit by going to T. This causes all the saved words except LBLOC to be restored and sends control to the second location after the RST which called it. A transfer to F is exactly the same, except that the input pointer is restored and control goes to the first location after the RST.

The addresses at the beginning of the recognizer define the local variables of the routine.

RSF recognize and skip if false

is also available.

Part of the user macro package is two macros for calling recognizers:

RBT A,B recognize and branch if true

calls recognizer A and branches to B if it returns true. If B is a number, it executes a CERRX CERRB instead of branching.

RBF recognize and branch if false

is the other macro.

During analysis, LBLOC points to the current element of the input line. The POP

CST compare and skip if true

compares this element with the word addressed and skips if they match. The complementary operation is

CSF compare and skip if false

The macros are

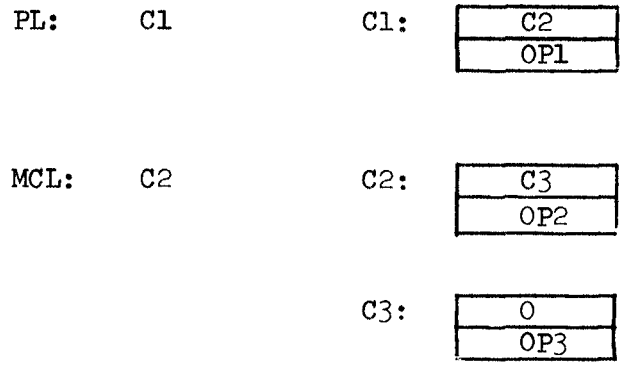
CBT and CBF

4.2 Compiling Code

Code is generated with a POP called CPL. CP maintains, in its standard form, two lists onto which code can be generated. These are headed by ICL and IPCL and consist of three word cells. The first word of each cell is either 0 or a pointer to the next word, the second is the compiled instruction, and the third is -1. Instructions are put onto a list with the POP CPL, whose action is illustrated in the figure on the next page. The instruction to be compiled is taken from the A register and put into a new list cell which is inserted just after the cell addressed by the argument of the POP.

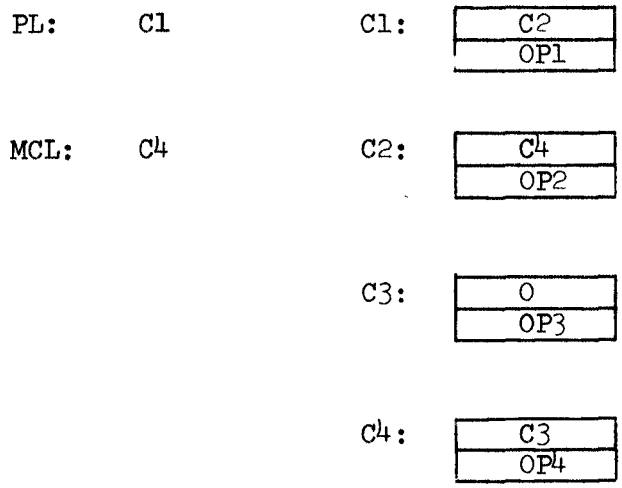
New list cells are assigned downward from the top of the compiler free storage area, while the recognizer stack grows up from the bottom. When they collide, the free storage area is expanded as described in the last section. Both lists and stack are reset at the beginning of each statement. The programmer therefore need not be concerned about losing track of list cells.

The fact that code is generated onto a list makes it easy to rearrange it and to maintain complex relationships between the order of



(a) Program list and pointer before execution of

```
LDA     OP4
CPL     MCL
```



(b) Program list and pointer after this CPL

FIGURE 1: Action of CPL

elements in the source line and the order of the code they generate.

A macro called C1 is in the user package.

```
C1      A,B
```

is equivalent to

```
LDA     ZA
MRG     B
CPL     MCL
```

That is, MCL is used as the pointer to the place at which code is currently being compiled. A may be thought of as the opcode, B as a word containing the address. It is assumed that the cell ZA has been constructed to hold the opcode A.

Also part of the user package are macros DP to define opcodes and DPS to create the Z-locations to hold them. Examples:

```
DP      ADD,55,SUB,54
DPS     ADD,SUB
```

In order to facilitate relative addressing within the code, a gimmick has been put into CPL. When this POP receives an instruction whose address points to a cell on a program list, it changes that address to 0. The address of the instruction in the cell pointed to is changed to the address of the cell being constructed. That is, a backwards reference is changed into a forward reference. When the loop in CS which transcribes the program list sees such an address, it converts the forward reference into a relative forward address, i.e., into the difference between the final locations of the instruction with the forward reference and the one being referred to. This computation is done with the subroutine COUNT, which returns the number of cells between the list cell addressed by A

and that addressed by B, plus 1, in A. Cells with second word =-1 are not counted.

The user must provide a bit table with one bit for each of the 128 possible opcodes. If a bit in this table is on, the corresponding opcode will be exempted from this test for relative reference. The address of the first word of the table should be RRTAB; it will occupy 6 words.

CPL and CS will recognize an address as referring to the program list if it is bigger than the contents of EPROG and if the second following word is -1. This arrangement is successful because CPL puts -1 into the second value word of the cell it compiles.

To create new lists, the user macro INIT is available. INIT M makes a new cell, clears its contents to 0, and puts its address in M.

To insert an entire list into another one, the POP MERGE is available. MERGE M inserts the list whose first cell is addressed by A after the cell addressed by M. M is changed to point to the last cell of the inserted list. A CPL is equivalent to an INIT, followed by an explicit store of the binary word to be compiled, followed by a MERGE.

To sequence down a list, the POP LVDI (load value double and increment) is available. LVDI M returns without skipping if M points to the last cell of a list. Otherwise it changes M to point to the next cell, puts the second and third words of this all into AB, and skips.

4.3 The Compiler Loop

The main loop of the compiler starts at CS. It does the following things:

a) calls RDL to read a line (section 6)

b) calls the user routine SETUP, which may do any initialization it likes

c) calls the pre-pass GNE

d) calls the user recognizer STAT, expecting it to return true

e) puts compiled code and text into program area unless the statement is direct (see section 10).

f) loops

Step (e) involves taking the instructions off the list ICL in order and adding them to the program area at the address in PLOC. If CODFLG is negative each instruction will be listed (see section 3.1).

4.4 Compiler Errors

When the compiler detects an error, it can call on standard error-handling machinery with the POP CERRX. If the input device is not the teletype, the offending statement is listed on the file LISTF. The address of the CERRX is then examined. If it is greater than TCFIM, it prints the message starting at the address, which should follow the conventions of TMSG (section 8). Otherwise, it prints the message at TCFIM (which is usually

THE CORRECT FORMAT IS)

and then proceeds as before.

After printing the error message, the POP returns to the main loop at CS.

If DEBUG is negative, the octal address of the CERRX is printed after the error message.

Refer to the discussion (in section 4.1) of numeric branch addresses for the recognize and compare macros.

5.0 Symbol Tables and Initialization

5.1 Structure of the Symbol Table

The symbol table in CP is a rather funny object which can be thought of as an elastic hash table. It consists of a fixed length table (between BST and EST) called the hash table, and a collection of five word cells attached to the hash table by pointers. Each symbol in the table belongs to one of these five word cells. The first word of the cell is a pointer. The next two words contain a string pointer to the symbol; the top 8 bits of each of these words are available to the user. The last two words are value words and completely at the user's disposal. The symbol table may be thought of as a device for associating with a string an address called the index. This address is always the address of the first value word; this number is returned by the lookup routine.

Each word of the hash table is either 0 or a pointer to the head of a list of these five word cells. When a symbol is looked up, a number less than the length of the hash table is computed from the string. This number is called a hash code. The word of the hash table corresponding to the hash code is picked up and the list which it heads is searched for the symbol. If a string identical to the one being looked up is found on this list, then the symbol is already in the table and its index can be returned. Otherwise a new cell must be added to the list and initialized with a pointer to the string and 0 value.

5.2 Lookup and Insertion Routines

To look up a symbol, put a string pointer to it in AB and call LKUPN. If the symbol is in the table, LKUPN skips and returns the index in X. Otherwise it returns without a skip and with the string pointer still in AB.

To insert a symbol, call INSN immediately after an unsuccessful call of LKUPN without disturbing the central registers. INSN returns the index in X, just as LKUPN does.

5.3 Initialization

The initialization routine GIN performs a number of functions connected with storage allocation, for which see section 2. It also calls IST, which initializes the symbol table as described in section 3.2.

6.0 Input

The routine RDL reads one line into a source line buffer called SBUF which has room for 300 characters. It expects to find a string pointer to the old contents of SBUF in SIP and leaves a pointer to the new contents in SIP and also in IP.

RDL reads from the file specified by INFIL. If this is not the teletype, no output is generated by RDL. Multiple blanks are correctly converted, and an end of file character sets INFIL back to the teletype. A line feed is assumed to be followed by a carriage return and line feed; the two characters following a line feed are therefore ignored. A carriage return is assumed to be followed by a line feed. This character, and all characters intervening betweer it and the preceding carriage return, are ignored.

If the teletype is the input medium, all the facilities of the QED line edit are made available by RDL. The line being edited is assumed to be pointed to by SIP. This means that, unless the user takes special action, the line being edited will always be the one previously typed in. "Line" refers of course to the logical line, i.e., the string of characters preceding the first carriage return (or D^c or F^c). Line feed is interpreted as a continuation character and causes carriage return and line feed to be printed. Q^c has been modified so that it deletes one physical line at each application.

7.0 Output

The system generates line feeds only in a routine called CRLF, which is responsible for printing carriage return and line feed and for advancing to a new page if necessary ("printing" here and below means "writing on OUTFIL"). The line on the current page is kept track of in LINCNT, which starts at 55. The page number is kept in PAGENO. If the symbol PAGES is >0 during assembly of CP, GIN will request from the user a page heading which it will put in HBUF. A pointer to the heading is kept in HEAD. The heading and the page number will be printed at the top of each page. If PAGES is <0, CRLF will simply generate carriage return and line feed.

A message output routine called TMSG is called with the (word) address of the message in X. It writes the characters in the message string onto OUTFIL until it sees a / and returns. It will print $ as carriage return and line feed.

A character output routine called CHOUT is identical to CIO OUTFIL except that it calls CRLF whenever it sees a carriage return or line feed.

8.0 Pre-Pass

If the symbol PREPAS is >0, the pre-pass routine GNE will be assembled into CP. When called, this routine calls the user initialization routine PPSET and then performs a rather elaborate analysis of the input line, which it finds a string pointer to in IP. The result is a series of integers starting at LBUF. The last of these integers is pointed to by LBLOC when GNE returns, although CS resets LBLOC to LBUF and the CST and CSF POPs assume that it points to the element of the converted source line currently under consideration.

GNE works with a 64 word table called CHTAB which it indexes by the internal code for the character it is processing. Characters larger than 77_8 are regarded as illegal, except for line feed, which is ignored, and carriage return, which causes GNE to add IEOS to LBUF and return. Each one word entry in CHTAB specifies the treatment to be accorded a character. This word is organized in the following way:

Bits	Function
0-2	Indexes ICTAB for transfer on initial character if ≠0
3	Keep following blank
4	Keep preceding blank
5	Keep following blank if next character approves
6	Keep following blank if previous character approves
7	Illegal
8	Ignore
9	Unused
10-23	Take as internal identifier if < 4000. Otherwise transfer to this address

If the first three bits of the CHTAB entry for the first character of the line are non-zero, GNE does an indexed branch to ICTAB.

There is a macro called CT which is part of the user package.

This macro is convenient for constructing the character table used by GNE. Its arguments have the following significance (avoid omitting them in the middle, since Arpas does not like this)

1 Bits 0-23. May be external. If /, ignore. If *, illegal

2 Blank treatment. Two characters, first for preceding blank, second for following. K=keep, C-conditionally keep, N=don't keep.

3 Initial code

GNE calls a routine called CHAR to get its characters from the source string, which is pointed to by IP. This routine just does a GCI, returning a carriage return if the string is exhausted. GNE puts the character it is working on into NC and keeps the preceding character in CC.

The user can cause processing of certain characters to cause transfers out of GNE by putting the transfer addresses into CHTAB. There are several standard points where he can re-enter GNE:

GNE11 to process a name. The first character should have been read (i.e., should be in NC). GNE will collect letters, digits and dots for as long as possible, look up the name and put its index into LBUF.

GNE12 to process a number. The first character should have been read. GNE will collect digits, one dot, and one E, possibly followed by a + or - and more digits. It will call a user routine, PPNL, with a string pointer to the number in AB. This routine should return an identification for the number, which will be put in LBUF. GNE will recover the string storage used for the number.

GNE1 to start a new line - PPSET will be called again. Everything will be reset and processing will continue.

GNE3 for main loop. The next character should be in NC.

GNE7 with an internal identifier in A. It will be stored and CHAR called for the next character.

GNE8 to call CHAR for the next character and loop.

When the pre-pass needs to write characters into string storage, it uses a POP called WC, which acts like WCI but takes no address. The address given in CSS is incremented by 1 and the character is written there.

9.0 Panic Control

If control is transferred to RUBOUT, the system will clear all input-output, reset the input and output files to teletype and the echo table to 2, wait for the user to type a character, and start up a fork in which RDL is called from the main loop at CS.

If a panic occurs out of this fork, it is checked for type. Illegal instruction traps print

ILLEGAL INSTRUCTION EXECUTED AT XXXX

and transfer to IITRP. Memory traps print

MEMORY TRAP

and transfer to MEMTRP. A BRS 10 causes the fork to be restarted as above after resetting the echo table to 2 (break on control characters only) but without any other initialization.

A rubout causes RUNFLG to be checked. If it is -1, it is reduced to -2, PANFLG is set to -1, and the program is restarted. If RUNFLG is -2, control goes to RUBOUT. If it is positive and the program is not waiting for I/O, the same action is taken as for RUNFLG = -1. If the program is waiting for I/O, CCLR is called and control goes to RUBOUT.

The idea behind all this is that RUNFLG is positive during compilation and -1 while the program is running. A running program or compile ought not to be dismissed, since it may have pointers in a bad state. Hence, PANFLG is set. It can be checked at the next convenient point in the compiler or running program. In particular, it is checked by CS after each statement is compiled. However, if the compiler is waiting for I/O or if two rubouts have occurred while the program is running, it seems desirable to interrupt.

The panic table starts at FT. FTA, FTB and FTX refer to the central register locations, FTM to the location indicating the status of the fork.

To use the program without the fork, change RBOUT3 from BRS 9 to BRU DSTRT.

10.0 The Executive

The CP executive maintains the source and object program. It accepts commands which allow the user to change the program. More commands can be added.

A CP command can be preceded by 0, 1 or 2 arguments. Each argument is the address of a logical line (string of characters bounded by carriage returns) in the source program. This address is formed out of a base and any number of displacements. The base may be

- `.` referring to the current line
- `:string:` referring to the first line after the current one which begins with the specified string followed by a character which is not a letter or digit. The source program is regarded as a ring for this search.
- `[string]` is like the : construction except that any occurrence of the string will do.

The displacements must be decimal integers. They are separated by +, -, or spaces (equivalent to +). The arguments are separated by commas.

Each argument, once recognized, occupies five words in ARGBUF. The first two words are a string pointer to the text of the statement, the next two a string pointer to the label of the nearest preceding labeled statement, and the last the distance of this statement. A routine called FTARG generates an error if there are no arguments and copies a single argument to make two.

The user must supply a routine called FSTAT which, called with a string pointer in AB and a number in X, will return in AB the first and last cells occupied by the code for the statement specified by the given label and displacement.

Commands are specified by a command table which the user must provide. The part of the command table built into CP is listed in the CP temporary

storage package. The table begins at BCT and ends just before ECT. It must be alphabetized. Each entry has the form:

several words containing the characters of the command, the last filled out by 200 characters.

a word with the sign bit set containing the address to go to when the command is recognized.

When a command is recognized, which occurs when enough characters have been typed to identify it, the remainder of the command is types out unless QCKFLG is negative. There is a macro called CD for defining commands. Its use is illustrated by the definitions of the built-in commands.

The built-in commands and their functions are as follows:

APPEND	0 args	starts input at the end of the current program
CHANGE	1 or 2 args	deletes the specified lines and accepts input to replace them
CODE	0 args	causes the code produced by the compiler to be listed
DELETE	1 or 2 args	deletes the specified lines
EDIT	1 arg	types the specified line, deletes it, makes it the line being edited and accepts input to replace it
INSERT	1 arg	accepts input to go before the specified line
MODIFY	1 arg	like edit but does not type
NOCODE	0 args	inverts CODE
PRINT	0,1 or 2 args	prints the specified part of the source program
QUICK	0 args	suppresses command completion
READ FROM	0,1 or 2 args	accepts input from a file specified after words and puts it at the end or before the specified statement
VERBOSE	0 args	inverts QUICK
WRITE	0,1 or 2 args	like PRINT, but on a specified file

/	0,1 or 2 args	like PRINT
↑	0 args	print preceding line
line feed	0 args	print next line

The routine OKTOGO is available to require that the next character be a dot, and to output a carriage return after it.

The label CPEXER is the error exit for the exec. It prints ? and goes to get another command.

The entry point to the executive is EXEC. This is the place to go when the compiled program returns from execution and wants to return control to the executive.

A very useful routine for copying strings is built into CP. The sequence

```
LDA     =PTR1
LDB     =PTR2
SBRM    COPYST
```

will cause the string specified by the pointer at PTR1 to be copied onto the end of the string in PTR2. The second word of PTR2 will be increased by the appropriate amount. The two words after the last word copied into will be destroyed. If the strings overlap, only downward copying will work, since the copying is done from the beginning of the string to the end.

This routine moves the strings word by word, using shifts and masks. It is about 20 times as fast as a GCI-WCI loop.

The executive interprets all input (except after INSERT, CHANGE, APPEND, EDIT and MODIFY commands) as commands with preceding arguments, except where the first character is a blank. In this case the input is taken to be a direct statement, which is analyzed and compiled. The compiled code is put into the input buffer, starting at LBUF. When the statement has been compiled, control

goes to the address XDS which must be supplied by the user.

When input is being accepted after an INSERT, CHANGE, APPEND, EDIT or MODIFY command, each line (terminated by carriage return) is taken to be a statement which is immediately compiled. Any error gives rise to an error comment and prevents code from being generated. During input of each line, the line last input, or the one just designated by an EDIT or MODIFY command, is being edited, and all QED control characters apply, with the exception that an initial D^c indicates the end of input and sends control back to the exec. This is the only way, aside from rubout, to terminate input. Input after an EDIT or MODIFY is not automatically restricted to one statement.

APPENDIX A: CP Parameters -- Symbols which must be provided by the user.

NAME	SECTION	TYPE	DESCRIPTION
NOSYMS	2.0	V	Number of symbols
LSS	2.0	V	Length of string storage
ILCFS	2.0	V	Initial length of compiler free storage
TOP	2.0	V	Top of available memory
LPROG	2.0	V	Initial length of program area
BFS	2.0	V	Bottom of available memory
INS	3.2	String	Initialization for symbol table
SETUP	3.2	S	Routine called before compiling each source line
STIDX	3.2	T	Table for storing addresses of strings being initialized
CHTAB	8.0	T	Character table for pre-pass
PPSET	8.0	S	Routine called before pre-pass on each source line
ICTAB	8.0	T	Initial character table for pre-pass
PPNL	8.0	S	Routine called with each number by pre-pass
ILCHAR	8.0	A	Go here on illegal character
IEOS	8.0	V	Internal identifier for end of statement
APS	2.0	S	Assign permanent storage
RPROG	2.0	S	Relocate program
MSPACE	2.0	S	Make space available
GC	2.0	S	String storage garbage collector
FSTAT	10.0	S	Find statement
BCT	10.0	T	Beginning of command table
ECT	10.0	T	End of command table
TCFIM	4.4	T	Dividing address for error messages
BLCHAR	3.1	V	Character typed by exec
OPTAB	3.1	T	Table of mnemonics for opcodes
POPTAB	3.1	T	Table of mnemonics for popcodes
OVFLOW	3.4	S	Called if GC returns too little free storage
IITRP	3.4	A	Go here on illegal instruction trap
MEMTRP	3.4	A	Go here on memory trap
RERR	3.4	S	Called on unexpected errors
XDS	10.0	A	Go here to execute direct statement
RRTAB	4.2	T	Bit table to suppress relative reference checking

APPENDIX B: Symbols provided by CP to the user.

NAME	SECTION	TYPE	DESCRIPTION
RST	4.1	P	Recognize and skip if true
RSF	4.1	P	Recognize and skip if false
CST	4.1	P	Compare and skip if true
CSF	4.1	P	Compare and skip if false
CPL	4.2	P	Compile instruction
MERGE	4.2	P	Merge one program list into another
LVDI	4.2	P	Sequence down program list
WC	8.0	P	Write character on string storage
CERRX	4.3	P	Compiler error
LKUPN	5.2	S	Look up name
INSN	5.2	S	Insert name
BST	5.1	T	Beginning of symbol table
EST	5.1	T	End of symbol table
GIN	3.2	S	General initialization
FSTART	3.8	A	First start; transcribe POP transfer vector
RDL	6.0	S	Read a line
CHAR	8.0	S	Get a character for pre-pass
GNE	8.0	S	The pre-pass routine
GNE1	8.0	A	To start over on a new line
GNE3	8.0	A	Main loop of pre-pass
GNE7	8.0	A	Return internal identifier to pre-pass
GNE8	8.0	A	Call CHAR and go to main loop
GNE11	8.0	A	Process name
GNE12	8.0	A	Process number
LBLOC	4.1	V	Input pointer
LBUF	8.0	T	Input buffer
SBUF	6.0	T	Source line buffer
T	4.1	A	True return from recognizer
F	4.1	A	False return from recognizer
COUNT	4.2	S	Count instructions in program list
INDL	4.2	S	Initialize a program list
PLOC	4.2	V	Location counter in program buffer
ICL	4.2	V	Initial word of the standard program list
IPCL	4.2	V	Initial word of another standard program list
MCL	4.2	V	Starts equal to ICL

NAME	SECTION	TYPE	DESCRIPTION
CS	4.2	A	Beginning of compiler loop
CSA	4.2	A	Print carriage return, then go to CS
RUBOUT	9.0	A	Set up fork and go to exec
DSTRT	9.0	A	Address to simulate BRS 9 for debugging
FT	9.0	V	First word of fork table (program counter)
FTA	9.0	V	Second word of fork table (A register)
FTB	9.0	V	Third word of fork table (B register)
FTX	9.0	V	Fourth word of fork table (X register)
FTM	9.0	V	Seventh word of fork table (status)
CHOUT	7.0	S	Print character, treating cf and lf specially
CRLF	7.0	S	Print carriage return and line feed. Paginate if necessary
TMSG	7.0	S	Type message
INFIL	6.0	V	Input file
OUTFIL	7.0	V	Output file
LISTF	3.1	V	Listing file
LINCNT	7.0	V	Line count on this page
PAGENO	7.0	V	Page number
HBUF	7.0	T	Heading buffer
EXEC	10.0	A	Entry point to exec
CPEXER	10.0	A	CP exec error entry
BPROG	2.0	T	Beginning of program area
EPROG	2.0	T	End of program area
CEPROG	2.0	T	Current end of program
BSTS	2.0	T	Beginning of source program (character address)
ESTS	2.0	T	End of source program (character address)
ARGBUF	2.0	T	Argument buffer
DEL	10.0	S	Delete subroutine
FTARG	10.0	S	Force two arguments
OKTOGO	10.0	S	Wait for dot
COPYST	10.0	S	Copy string
PANFLG	9.0	V	Panic (rubout) flag
RUNFLG	9.0	V	Program running flag
DEBUG	10.0	V	Debug flag
BRFBEG	3.2	V	Brief beginning flag
CODFLG	3.1	V	Code listing flag

Types:

A	address to which control may be transferred
P	POP
S	subroutine called with SBRM
T	table
V	single word whose value is of concern