The BCPL Cintcode System Users' Guide

by

Martin Richards

mr@cl.cam.ac.uk

http://www.cl.cam.ac.uk/~mr/

Computer Laboratory
University of Cambridge

Archive version: December 11, 2003

Abstract

BCPL is a simple systems programming language with a small fast compiler which is easily ported to new machines. The language was first implemented in 1967 and has been in continuous use since then. Its notable features typelessness and the ability to provide machine independent pointer arithmetic allowing a simple way to represent vectors and structures. BCPL functions are recursive and variadic but, like C, do not allow dynamic free variables, and so can be represented by just their entry addresses. There is no built-in garbage collector and all input-output is done using library calls.

This document describes the BCPL Cintcode System giving a definition of the language, its library and running environment. It also describes how to obtain and install the system on your own machine.

Keywords

Systems programming language, Typeless language, BCPL, Cintcode, Coroutines.

Contents

Preface

The concept for BCPL originated in 1966 and was first outlined in my PhD thesis [Ric66] later that year. Its first implementation had to wait until 1967 when I was working at M.I.T. Its heyday was perhaps from the mid 70s to the mid 80s, but even now it is still continues to be used at some universities, in industry and by private individuals. It is a useful language for experimenting with algorithms and for research in optimizing compilers. As might be expected it is beginning to show its age and is unlikely to be able to compete with modern current day languages or even C, although in specialist applications where small size, simplicity and portability are important it may still have a place.

This report is intended to provide a record of the main features of the BCPL in sufficient depth so that a serious reader may obtain a proper understanding of philosophy behind the design and use of the language. A complete interpretive implementation is presented the source of which is freely available from my home page on the Internet [Ric].

The implementation is reasonably machine independent and will run efficiently on many machines both now and in the indefinite future.

Some of the more significant topics covered by this report are listed below.

- A specification of the BCPL.
- A description of the runtime library.
- The design and implementation of command language interpreter for the system.
- The design of its portable implementation using the intermediate object code OCODE.
- The design of Cintcode, the compact byte stream target code used in this implementation.
- A description of the BCPL Cintcode debugger.

- The efficient implementation of the Cintcode interpreter for several processors including both RISC and i386/Pentium based machines.
- The profiling and statistics gathering facilities offered by the system.

Chapter 1

The System Overview

This book contains a full description of an interpretive implementation of BCPL that is supported by an command language and interactive low level debugger. As an introduction to the system an example console session is now presented to exhibit some of its key features.

1.1 A Console Session

When the system is started its opening message is as follows:

```
shep$ cinterp
BCPL Cintcode System
0>
```

The characters `0>` are followed by a space character and is the command language prompt string inviting the user to type a command. The integer gives the execution time of the preceeding command. A possible command allowing the user to type a BCPL program into the file `fact.b` is as follows:

```
> input fact.b
GET "libhdr"

LET fact(n) = n=0 -> 1, n*fact(n-1)

AND start() = VALOF
{ FOR i = 1 TO 5 DO writef("fact(%n) = %i5*n", i, fact(i))
   RESULTIS 0
}
/*
10>
```

In this program, the directive `GET "libhdr"` causes a file of standard library declarations to be inserted at that position. The text

```
LET start() = VALOF
```

is the heading for the declaration of the function `start` which, by convention, is the first function to be called when a program is run. The empty parentheses `()` indicate that the routine takes no arguments. The text

```
FOR i = 1 TO 5 DO
```

introduces a for-loop whose control variable `i` successively takes the values from `1` to `5`. The body of the for-loop is a call of the library routine `writef` whose effect is to output the format string after replacing the format items `%n` and `%i5` by appropriately formatted representations of `i` and `fact(i)`. Within the string `*n` represents the newline character. The statement `RESULTIS 0` causes a return from the `VALOF` contruct with the value `0` which become the result of `start`. This value indicates that the program successfully completed. The text

```
AND fact(n) =
```

introduces the definition of the function `fact` which take one argument (`n`) and yields `n` factorial. This program can be compiled by using the following command:

```
10> bcpl fact.b to fact

BCPL (14 Dec 1999)
Code size = 96 bytes
10>
```

This command compiles the source file `fact.b` creating an executable object module in the file called `fact`. The program can then be run by simply typing the name of this file.

```
10> fact
fact(1) =     1
fact(2) =     2
fact(3) =     6
fact(4) =    24
fact(5) =   120
0>
```

The system provides several commands including a text editor `edit` and various file utilities such as `join`, `rename` and `delete`. These are described in chapter 7, and will not be demonstrated here. Instead, this console session will be completed by exhibiting some of the features of the undelying interpretive code Cintcode and the interactive debugger.

When the BCPL compiler is invoked, it can be given additional arguments that control various compiler options. One of these is the `d1` option that directs the compiler to output the compiled code in a readable form for the user to inspect. Using this option on `fact.b` the following information is generated.

```
10> bcpl fact.b to fact d1

BCPL (14 Dec 1999)
   0:  DATAW 0x00000000
   4:  DATAW 0x0000DFDF
   8:  DATAW 0x63616607
  12:  DATAW 0x20202074
// Entry to:   fact
  16: L1:
  16:   JNE0  L4
  18:     L1
  19:    RTN
  20: L4:
  20:    LM1
  21:    AP3
  22:     LF  L1
  24:     K4
  25:    LP3
  26:    MUL
  27:    RTN
  28:  DATAW 0x0000DFDF
  32:  DATAW 0x61747307
  36:  DATAW 0x20207472
// Entry to:   start
  40: L2:
  40:     L1
  41:    SP3
  42: L5:
  42:    LP3
  43:     LF  L1
  45:     K9
  46:    SP9
  47:    LP3
  48:    SP8
  49:    LLL  L2920
  51:    K4G   70
  53:     L1
  54:    AP3
  55:    SP3
  56:     L5
  57:    JLE  L5
  59:     L0
  60:    RTN
  64: L2920:
  64:  DATAW 0x6361660F
  68:  DATAW 0x6E252874
  72:  DATAW 0x203D2029
  76:  DATAW 0x0A356925
  80: L3:
  80:  DATAW 0x00000000
  84:  DATAW 0x00000001
  88:  DATAW 0x00000028
  92:  DATAW 0x00000046
Program size = 96 bytes
20>
```

This output shows the sequence of CINTCODE instructions compiled for the

two procedures defined in the factorial program. In addition to the instructions, there are a number of data words holding a string constant, initialisation data and symbolic information for the debugger. The data word at location 4 holds a special bit pattern indicating the presence of a procedure name placed just before the procedure entry point. As can be seen the procedure in this case is `fact`. Similar information is packed at location 28 for the function `start`. Most Cintcode instructions occupy one byte and perform simple operations on the registers and memory of the Cintcode machine. For instance, the first two instructions of `start` (`L1` and `SP3` at locations 40 and 41) load the constant `1` into a register called `A` and then stores it at offset 3 relative to the stack frame pointer (called `P`). This corresponds to the initialisation of the for-loop control variable `i`. The label `L5` marks the start of the for-loop body where `fact(i)` is first computed (`LP3` loads `i`, `LF L1` loads the entry address of `fact` and `K9` makes the function call incrementing the `P` pointer by `9` locations). The result of this function is returned in `A` which is then placed in the stack by means of `SP9` in the appropriate position for the third argument of the call of `writef` that is currently being setup. The second argument, `i`, is setup by means of `LP3 SP8`, and the first argument which is the format string is loaded by `LLL L2920`. The next instruction (`K4G 70`) causes the routine `writef`, whose entry point is in global variable 70, to be called incrementing the stack pointer by 4 words as it does so. Thus the compilation of the call `writef("fact(%n) = %i5*n", i, f(i))` occupies just 11 bytes from location 42 to 52, plus the 16 bytes at location 64 where the format string is packed. The next three instructions (`L1 AP3 SP3`) increment `i`. To test that `i` is still less than `5`, the value `5` is loaded (`L5`) and a conditional jump instruction (`JNE L5`) obeyed, if the jump is not taken control falls through to the instruction `L0 RTN` causing a return from the routine with result `0`. As can be seen, each instruction of this routine occupy one byte except for the `LF`, `LLL`, `K4G` and `JNE` instructions which each occupy two bytes. The body of the function `fact` is even more compact and equally easy to understand. It starts by testing whether its argument is zero (`JNE0 L4`). If it is, it loads `1` (`L1`) and returns (`RTN`). If not, it computes `n-1` by loading `-1` (`LM1`) and adding `n` (`AP3`) before loading the function `fact` (`LF L1`) and calling it (`K4`). The result of this call is multiplied by `n` (`LP3 MUL`) before returning from `fact` (`RTN`). The space occupied by this code is just 12 bytes.

The debugger can be entered using the abort command.

```
20> abort

!! ABORT 99: User requested
*
```

The asterisk is the prompt inviting the user to enter a debugging command. The debugger provides facilities for inspecting and changing memory as well as setting breakpoints and performing single step execution of the program. As an example, a breakpoint is placed at the first instruction of the routine `clihook` which is used by the CLI to transfer control to a command. Consider the following commands:

```
* g4 b1
* b
1:        clihook
*
```

This first loads the entry point of `clihook` (`g4`) since it happens to be held in global variable 4 and sets (`b1`) breakpoint number 1 at this position. The command `b`, without an argument, lists the current breakpoints confirming that the correct one has been set. Normal execution can continue using the `c` command.

```
* c
0>
```

If we now try to execute the factorial program, we immediately hit the breakpoint.

```
0> fact

!! BPT 1  clihook
   A=          0 B=          0     3340:    K4G  1
*
```

This indicates that the breakpoint occured when the Cintcode registera `A` and `B` were both zero, and that the program counter is set to 3300 where the next instruction to be obeyed is `K6G 1`. Single Cintcode instructions can now be executed using the `\` command.

```
* \A=          0 B=          0    24208:     L1
* \A=          1 B=          0    24209:    SP3
* \A=          1 B=          0    24210:    LP3
*
```

After each single step execution a summary of the current state is printed. In the above sequence we see that the execution of the instruction `L1` loading `1` into the `A` register. The execution of `SP3` does not have an immediately observable effect since it updates a local variable held in the current stack frame, but the stack frame can be displayed using the `t` command.

```
* p t4

P  0:      24292        3342     start             1
*
```

This confirms that location `P3` contains the value `1` corresponding to the initial value of the for-loop control variable `i`. At this stage it is possible to change its value to `3`, say.

```
* 3 sp3
* p t4

P  0:      24092        3302     start             3
*
```

If single stepping is continued for a while we observe the evaluation of the recursive call `fact(3)`.

```
* \A=          3 B=          1    24211:     LF  24184
* \A=    fact    B=          3    24213:     K9
* \A=          3 B=          3    24184:   JNE0  24188
* \A=          3 B=          3    24188:    LM1
* \A=         -1 B=          3    24189:    AP3
* \A=          2 B=          3    24190:     LF  24184
* \A=    fact    B=          2    24192:     K4
* \A=          2 B=          2    24184:   JNE0  24188
* \A=          2 B=          2    24188:    LM1
* \A=         -1 B=          2    24189:    AP3
* \A=          1 B=          2    24190:     LF  24184
* \A=    fact    B=          1    24192:     K4
* \A=          1 B=          1    24184:   JNE0  24188
* \A=          1 B=          1    24188:    LM1
* \A=         -1 B=          1    24189:    AP3
* \A=          0 B=          1    24190:     LF  24184
* \A=    fact    B=          0    24192:     K4
* \A=          0 B=          0    24184:   JNE0  24188
* \A=          0 B=          0    24186:     L1
* \A=          1 B=          0    24187:    RTN
* \A=          1 B=          0    24193:    LP3
* \A=          1 B=          1    24194:    MUL
* \A=          1 B=          1    24195:    RTN
* \A=          1 B=          1    24193:    LP3
* \A=          2 B=          1    24194:    MUL
* \A=          2 B=          1    24195:    RTN
* \A=          2 B=          1    24193:    LP3
* \A=          3 B=          2    24194:    MUL
* \A=          6 B=          2    24195:    RTN
* \A=          6 B=          2    24214:    SP9
* \A=          6 B=          2    24215:    LP3
* \A=          3 B=          6    24216:    SP8
* \A=          3 B=          6    24217:    LLL  24232
* \A=       6058 B=          3    24219:    K4G  70
*
```

At this moment the routine `writef` is just about to be entered to print an message about factorial 3. We can unset breakpoint 1 and continue normal execution by typing `0b1 c`.

```
* 0b1 c
fact(1) =     1
fact(4) =    24
fact(5) =   120
30>
```

Notice that `fact(1)` is the first to be written since it has already been evaluated, but the next time round the `FOR` loop `i` has value `4` since `i` was updated during the debugging session.

As one final example in this session we will re-compile the BCPL compiler.

```
30> bcpl com/bcpl.b to junk

BCPL (14 Dec 1999)
Code size = 8576 bytes
Code size = 5460 bytes
Code size = 12196 bytes
800>
```

This shows that the total size of the compiler is 26,232 bytes and that it can be compiled (on a 450MHz Pentium machine) in 0.8 seconds. Since this involves executing 17,727,220 cintcode instructions, the rate is just over 22 million instructions per second with the current interpreter.

Chapter 2

The BCPL Language

The design of BCPL was strongly influenced by the work done jointly by Cambridge and London Universities on a language called CPL (Combined Programming Language) which was conceived at Cambridge to be the main language to run on the new and powerful Ferranti Atlas computer to be installed back in 1963. Another Atlas computer was installed in London at about the same time and it was decided to make the development of CPL a joint project between Cambridge and London Universities. As a result the name changed from Cambridge Programming Language to Combined Programming Language. It could reasonably be called Christopher's Programming Language in recognition of Christpher Strachey whose bubbling enthusiasm and talent steered the course of its development.

CPL was an ambitious language in the ALGOL tradition but with many novel and significant extensions intended to make its realm of application more general. These included a greater richness in control constructs such as the now well known `IF`, `UNLESS`, `WHILE`, `UNTIL`, `REPEATWHILE`, `SWITCHON` statements. It could handle a wide variety of data types including string and bit patterns and was one of the first strictly typed languages to provided a structure mechanism that permitted convenient handling of lists, trees and directed graphs. Work on CPL ran from about 1961 to 1967, but was hampered by a number of factors that eventually killed it. It was, for instance, too large and complicated for the machines available at the time, and the desire for elegance and mathematical cleanliness outweighed the more pragmatic arguments for efficiency and implementability. Much of the implementation was done by research students who came and left during the lifetime of the project. As soon as they knew enough to be useful they had to transfer their attention to writing their theses. Another problem (that became of particular interest to me) was that the implementation had to move from EDSAC II to the Atlas computer about halfway through the project. The CPL compiler

was thus required to be portable. This was achieved by writing it in a simple subset of CPL which was then hand translated into a sequence of low level macro calls that could be expanded into the assembly language of either machine by selecting an appropriate set of macro definitions. The macrogenerator used was GPM[Str65] which was designed specifically for this task by Strachey. It was a delightfully elegant work of art in its own right and is well worth study. A variant of GPM, called BGPM, is included in the standard BCPL distribution.

BCPL was initially similar to the subset of CPL used in the encoding of the CPL compiler. An outline of BCPL's main features first appeared in my PhD thesis [Ric66] in 1966 but it was not fully designed and implemented until early the following year when I was working at Project MAC of the Massachussetts Institute of Technology. Its first implementation was written in Doug Ross's Algol Extended for Design (AED-0)[D.T64] which was the only language then available on CTSS, the time sharing system at Project MAC, other than LISP that allowed recursion.

2.1 Language Overview

A BCPL program is made up of one or more separately compiled sections, each consisting of a list of declarations that define the constants, static data and functions belonging to the section. Within functions it is possible to declare dynamic variables and vectors that exist only as long as they are required. The language is designed so that these dynamic quantities can be allocated space on a simple runtime stack. The addressing of these quantities is relative to the base of the stack frame belonging to the current function activation. For this to be efficient, dynamic vectors have sizes that are known at compile time. Functions may be called recursively and their arguments are called by value. The effect of call by reference can be achieved by passing pointers. Input and output and other system operations are provided by means of library functions.

The main syntactic components of BCPL are: expressions, commands, and declarations. These are described in the next few sections. In general, the purpose of an expression is to compute a value, while the purpose of a command is normally to change the value of one or more variables.

2.1.1 Comments

There are two form of comments. One starts with the symbol `//` and extends up to but not including the end-of-line character, and the other starts with the symbol `/*` and ends at a matching occurrence of `*/`. Comment brackets (`/*`

and */ may be nested, and within such a comments the lexical analyser is only looking for /* and */. Care is needed when commenting out fragments of program containing string constants. Comments are equivalent to white space and so may not occur in the middle of multi-character symbols such as identifiers or constants.

2.1.2 The GET Directive

A directives of the form GET "filename" is replaced by the contents of the named file. By convention, GET directives normally appear on separate lines.

2.1.3 Conditional Compilation

There is a simple mechanism, whose implementation takes fewer than 20 lines of code in the compiler's lexical analyser, that allow conditional skipping of lexical symbols. It uses directives of the following form:

$$*tag*
$<*tag*
$>*tag*

where *tag* is conditional compilation tag composed of letters, digits, dots and underlines. All tags are initially unset, but may be complemented using the $$*tag* directive. All the lexical tokens between $<*tag* and $>*tag* are skipped (treated as comments) unless the specified tag is set. The following example shows how this conditional compilation feature can be used.

```
$$Linux                    // Set the Linux conditional compilation tag

$<Linux                    // Include if the Linux tag is set
  $<WinNT $$WinNT $>WinNT  // Unset the WinNT tag if set
  writef("This was compiled for Linux")
$>Linux
$<WinNT                    // Include if the WinNT tag is set
  writef("This was compiled for Windows NT")
$>WinNT
```

2.1.4 Section Brackets

Historically BCPL used the symbols $(and $) to bracket commands and declarations. These symbols are called section brackets and are allowed to be followed by tags composed of letters, digits, dots and underlines. A tagged closing section bracket is forced to match with its corresponding open section bracket by the automatic insertion of extra closing brackets as needed. Use of this mechanism is no

longer recommended since it can lead to obscure programming errors. Recently BCPL has been extended to allow all untagged section brackets to be replaced by `{` and `}` as appropriate.

2.2 Expressions

Expressions are composed of names, constants and expression operators and may be grouped using parentheses. The precedence and associativity of the different expression constructs is given in Section 2.2.9. In the Cintcode implementation of BCPL all expressions yield values that are 32 bits long, but in some native code implementations this word length is 64 bits.

2.2.1 Names

Syntactically a name is of a sequence of letters, digits, dots and underlines starting with a letter that is not one of the reserved words (such as `IF`, `WHILE`, `TABLE`).

A name may be declared to a local variable, a static variable, a global variable, a manifest constant or a function. Since the language is typeless, the value of a name is a bit pattern whose interpretation depends on how it is used.

2.2.2 Constants

Decimal numbers consist of a sequence of digits, while binary, octal or hexadecimal hexadecimal are represented, repectively, by `#b`, `#o` or `#x` followed by digits of the appropriate sort. The `o` may be omitted in octal numbers. Underlines may be inserted within numbers to improve their readability. The following are examples of valid numbers:

```
1234
1_234_456
#b_1011_1100_0110
#o377
#x_BC6
```

The constants `TRUE` and `FALSE` have values `-1` and `0`, respectively, which are the conventional BCPL representations of the two truth values. Whenever a boolean test is made, the value is compared with with `FALSE` (`=0`).

A question mark (?) may be used as a constant with undefined value. It can be used in statements such as:

```
LET a, b, count = ?, ?, 0
sendpkt(P_notinuse, rdtask, ?, ?, Read, buf, size)
```

Character constants consist of a single character enclosed in single quotes ('). The character returns a value in the range 0 to 255 corresponding to its normal ASCII encoding.

Character (and string) constants may use the following escape sequences:

Escape	Replacement
*n	A newline (end-of-line) character.
*c	A carriage return character.
*p	A newpage (form-feed) character.
*s	A space character.
*b	A backspace character.
*t	A tab character.
*e	An escape character.
*"	"
*'	'
**	*
*x*hh*	The single character with number *hh* (two hexadecimal digits denoting an integer in the range [0,255]).
**ddd*	The single character with number *ddd* (three octal digits denoting an integer in the range [0,255]).
f..f	This sequence is ignored, where *f..f* stands for a sequence of one or more space, tab, newline and newpage characters.

A string constant consists of a sequence of zero or more characters enclosed within quotes ("). Both string and character constants use the same character escape mechanism described above. The value of a string is a pointer where the length and bytes of the string are packed. If s is a string then s%0 is its length and s%1 is its first character, see Section 2.2.6.

A static vector can be created using an expression of the following form: TABLE $K_0, \ldots, K_n$ where $K_0, \ldots, K_n$ are manifest constant expressions, see Section 2.2.10. The space for a static vector is allocated for the lifetime of the program and its elements are updatable.

2.2.3 Calls

The only difference between functions and routines is whether their calls return results. Functions calls are normally made in the context of an expression where a result is required, while routine calls are made in the context of a command no requiring a result. If a function is called in the context of a command its result is thrown away, and if a routine is called in the context of an expression the result is undefined.

A call is syntactically an expression followed by a list of arguments enclosed in paretheses.

```
newline()
mk3(Mult, x, y)
writef("f(%n) = %n*n", i, f(i))
f(1,2,3)
(fntab!i)(p, @a)
```

The parentheses are required even if no arguments are given. The last example above illustrates a call in which the function is specified by an expression. Section 2.4.8 covers both procedure definition and procedure calls.

2.2.4 Method Calls

Method calls are designed to make an object oriented style of programming more convenient. They are syntactically similar to a function calls but uses a hash symbol (`#`) to separate the function specifier from its arguments. The expression:

$$E\texttt{\#(}E_1\texttt{,..,}E_n\texttt{)}$$

is defined to be equivalent to:

$$\texttt{(}E_1\texttt{!0!}E\texttt{)(}E_1\texttt{,..,}E_n\texttt{)}$$

Here, E_1 points to the fields of an object, with the convention that its zeroth field (E_1`!0`) is a pointer to the methods vector. Element E of this vector is applied to the given set of arguments. Normally, E is a manifest constant. An example program illustrating method calls can be found in `BCPL/bcplprogs/demos/objdemo.b` in the BCPL distribution system (see Chapter 8).

2.2.5 Prefixed Expression Operators

An expression of the form `!`E returns the contents of the memory word pointed to by the value of E.

An expression of the form `@`E returns a pointer to the word sized memory location specified by E. E can only be a variable name or an expression with leading operator `!`.

Expressions of the form: `+`E, `-`E, `ABS` E, `~`E and `NOT` E return the result of applying the given prefixed operator to the value of the expression E. The operator `+` returns the value unchanged, `-` returns the integer negation, `ABS` returns the absolute value, `~` and `NOT` return the bitwise complement of the value. By convention, `~` is used for bit patterns and `NOT` for truth values.

Expressions of the form: `SLCT` *len*`:`*shift*`:`*offset* pack the three constants *len*, *shift* and *offset* into a word. Such packed constants are used by the field selection operator `OF` described in the next section.

`SLCT` *shift*`:`*offset* means `SLCT 0:`*shift*`:`*offset*, and `SLCT` *offset* means `SLCT 0:0:`*offset*.

2.2.6 Infixed Expression Operators

An expression of the form E_1`!`E_2 evaluates E_1 and E_2 to yield respectively a pointer, p say, and an integer, n say. The value returned is the contents of the n^{th} word relative to p.

An expression of the form E_1`%`E_2 evaluates E_1 and E_2 to yield a pointer, p say, and an integer, n say. The expression returns a word sized result equal to the unsigned byte at position n relative to p.

An expression of the form K `OF` E accesses a field of consecutive bits in memory. K must be a manifest constant (see section 2.2.10) equal to `SLCT` *len*`:`*shift*`:`*offset* and E must yield a pointer, p say. The field is contained entirely in the word at position p+*offset*. It has a bit length of *len* and is *shift* bits from the right hand end of the word. A length of zero is interpreted as the longest length possible consitent with *shift* and the word length of the implementation. The operator ⁖ is a synonym of `OF`. Both may be used on right and left hand side of assignments statements but not as the operand of `@`. When used in a right hand context the selected field is shifted to the right hand end of the result with vacated positions, if any, filled with zeros. A shift to the left is performed when a field is updated. Suppose `p!3` holds the value `#x12345678`, then after the assignment:

```
(SLCT 12:8:3) OF p := 1 + (SLCT 8:20:3) OF p
```

the value of `p!3` is `#x12302478`.

An expressions of the form E_1`<<`E_2 (or E_1`>>`E_2) evaluates E_1 and E_2 to yield a bit pattern, w say, and an integer, n say, and returns the result of shifting w to the left (or right) by n bit positions. Vacated positions are filled with zeroes. Negative shifts or ones of more than the word length return 0.

Expressions of the form: E_1`*`E_2, E_1`/`E_2, E_1 `REM` E_2, E_1`+`E_2, E_1`-`E_2. E_1 `EQV` E_2 and E_1 `NEQV` E_2 return the result of applying the given operator to the two operands. The operators are, respectively, integer multiplication, integer division, remainder after integer division, integer addition, integer subtraction, bitwise equivalent and bitwise not equivalent (exclusive OR). `MOD` and `XOR` can be used as synonyms of `REM` and `NEQV`, respectively.

Expressions of the form: E_1`&`E_2 and E_1`|`E_2 return, respectively, the bitwise AND or OR of their operands unless the expression is being evaluated in a boolean context such as the condition in a while command, in which case the operands are tested from from left to right until the value of the condition is known.

An expression of the form: *E relop E relop ... relop E* where each *relop* is one of `=`, `~=`, `<=`, `>=`, `<` or `>` returns `TRUE` if all the individual relations are satisfied and `FALSE`, otherwise. The operands are evaluated from left to right, and evaluation stops as soon as the result can be determined. Operands may be evaluated more than once, so don't try `'0'<=rdch()<='9'`.

An expression of the form: E_1`->`E_2`,`E_3 first evaluates E_1 in a boolean context, and, if this yields `FALSE`, it returns the value of E_3, otherwise it returns the value of E_2.

2.2.7 Boolean Evaluation

Expressions that control the flow of execution in coditional constructs, such as if and while commands, are evaluated in a Boolean. This effects the treatment of the operators `NOT`, `&` and `|`. In a Boolean context, the operands of `&` and `|` are evaluated from left to right until the value of the condition is know, and `NOT` (or `~`) negates the condition.

2.2.8 `VALOF` Expressions

An expression of the form `VALOF` C, where C is a command, is evaluated by executing the command C. On encountering a command of the form `RESULTIS` E within C, execution terminates, returning the value of E as the result of the `VALOF` expression. Valof expressions are often used as the bodies of functions.

2.2.9 Expression Precedence

So that the separator semicolon (`;`) can be omitted at the end of any line, there is the restriction that infixed operators may not occur as the first token of a line. So, if the first token on a line is `!`, `+` or `-`, these must be regarded as prefixed operators.

The syntax of BCPL is specified by the diagrams in Appendix A, but a summany of the precendence of expression operators is given in table 2.1. The precedence values are in the range 0 to 9, with the higher values signifying greater binding power. The letters L and R denote the associativity of the operators. For instance, the dyadic operator `-` is left associative and so `a-b-c` is equivalent to `(v-i)-j`, while `b1->x,b2->y,z` is right associative and so is equivalent to `b1->x,(b2->y,z)`.

9 9L	Names, Literals, `?`, `TRUE`, `FALSE`, `(E)`, Function and method calls	
8L 7	`!  %  OF  ::` `!  @`	Dyadic Prefixed
6L 5	`*  /  REM MOD` `+  -  ABS`	Dyadic operators
4	`=  ~=  <=  >=  <  >`	Extended Relations
4L	`<<  >>`	

3
3L
2L
1L
	`~ NOT`
	`&`
	`|`
	`EQV NEQV XOR`
		Bitwise and Boolean operators

1R	`-> ,`	Conditional expression
0 0	`VALOF TABLE` `SLCT :`	Valof and Table expressions Field selector constant

Table 2.1: Operator precedence

Notice that these precedence values imply that

```
            ! f x  means  ! (f x)
            ! @ x  means  ! (@ x)
      ! v ! i ! j  means  ! ((v!i)!j)
      @ v ! i ! j  means  @ ((v!i)!j)
    x << 1+y >> 1  means  (x<<(1+y))>>1)
            ~ x!y  means  ~ (x!y)
            ~ x=y  means  ~ (x=y)
          NOT x=y  means  NOT (x=y)
b1-> x, b2 -> y,z  means  b1 -> x, (b2 -> y, z)
```

2.2.10 Manifest Constant Expressions

Manifest constant expressions can be evaluated at compile time. They may only consist of manifest constant names, numbers and character constants, `TRUE`, `FALSE`, `?`, the operators `REM`,`SLCT`, `*`, `/`, `+`, `-`, `ABS`, the relational operators, `<<`, `>>`, `NOT`, `~`, `&`, `|`, `EQV`, `NEQV`, and conditional expressions. Manifest expressions are used in `MANIFEST`, `GLOBAL` and `STATIC` declarations, the upper bound in vector declarations and the step length in `FOR` commands, and as the left hand operand of `::` and `OF`.

2.3 Commands

The primary purpose of commands is for updating variables, for input/output operations, and for controlling the flow of control.

2.3.1 Assignments

A command of the form L`:=`E updates the location specified by the expression L with the value of expression E. The following are some examples:

```
cg_x := 1000
v!i := x+1
!ptr := mk3(op, a, b)
str%k := ch
%strp := 'A'
```

Syntactically, L must be either a variable name or an expression whose leading operator is `!` or `%`. If it is a name, it must have been declared as a static or dynamic variable. If the name denotes a function, it is only updatable if the function has been declared to reside in the global vector. If L has leading operator

!, then its evaluation (given in Section 2.2.6) leads to a memory location which is the one that is updated by the assignment. If the % operator is used, the appropriate 8 bit location is updated by the least significant 8 bits of E.

A multiple assignment has the following form:

$$L_1, \ldots, L_n \texttt{ := } E_1, \ldots, E_n$$

This construct allows a single command to make several assignments without needing to be enclosed in section brackets. The assignments are done from left and is eqivalent to:

$$L_1 \texttt{:=} E_1 \texttt{ ; } \ldots \texttt{ ; } L_n \texttt{ := } E_n$$

2.3.2 Calls

Both function calls and method calls as described in sections 2.2.3 and 2.2.4 are allowed to be executed as commands. The only difference is that any results produced are thrown away.

2.3.3 Conditional Commands

The syntax of the three conditional commands is as follows:

IF E DO C_1
UNLESS E DO C_2
TEST E THEN C_1 ELSE C_2

where E denotes an expression and C_1 and C_2 denote commands. The symbols DO and THEN may be omitted whenever they are followed by a command keyword. To execute a conditional command, the expression E is first evaluated. If it yields a non zero value and C_1 is present then C_1 is executed. If it yields zero and C_2 is present, C_2 is executed.

2.3.4 Repetitive Commands

The syntax of the repetitive commands is as follows:

WHILE E DO C
UNTIL E DO C
C REPEAT

C REPEATWHILE *E*
C REPEATUNTIL *E*
FOR *N* = E_1 TO E_2 DO *C*
FOR *N* = E_1 TO E_2 bY *K* DO *C*

The symbol DO may be omitted whenever it is followed by a command keyword. The WHILE command repeatedly executes the command *C* as long as *E* is non-zero. The UNTIL command executes *C* until *E* is zero. The REPEAT command executes *C* indefinitely. The REPEATWHILE and REPEATUNTIL commands first execute *C* then behave like WHILE *E* DO *C* or UNTIL *E* DO *C*, respectively.

The FOR command first initialises its control variable (*N*) to the value of E_1, and evaluates the end limit E_2. Until *N* moves beyond the end limit, the command *C* is executed and *N* increment by the step length given by *K* which must be a manifest constant expression (see Section 2.2.10). If BY *K* is omitted BY 1 is assumed. A FOR command starts a new dynamic scope and the control variable *N* is allocated a location within this new scope, as are all other dynamic variables and vectors within the FOR command.

2.3.5 SWITCHON command

A SWITCHON command has the following form:

SWITCHON *E* INTO { C_1 ; . . . ; C_n }

where the commands C_1 to C_n may have labels of the form DEFAULT: or CASE *K*. *E* is evaluated and then a jump is made to the place in the body labelled by the matching CASE label. If no CASE label with the required value exists, then control goes to the DEFAULT label if it exists, otherwise execution continues from just after the switch.

2.3.6 Flow of Control

The following commands affect the flow of control.

RESULTIS *E*
RETURN
ENDCASE
LOOP
BREAK
GOTO *E*
FINISH

`RESULTIS` causes evaluation of the smallest textually enclosing `VALOF` expression to return with the value of E.

`RETURN` causes evaluation of the current routine to terminate.

`LOOP` causes a jump to the point just after the end of the body of the smallest textually enclosing repetitive command (see Section 2.3.4). For a `REPEAT` command, this will cause the body to be executed again. For a `FOR` command, it causes a jump to where the control variable is incremented, and for the `REPEATWHILE` and `REPEATUNTIL` commands, it causes a jump to the place where the controlling expression is re-evaluated.

`BREAK` causes a jump to the point just after the smallest enclosing repetitive command (see Section 2.3.4).

`ENDCASE` causes execution of the commands in the smallest enclosing `SWITCHON` command to complete.

The `GOTO` command jumps to the command whose label is the value of E. See Section 2.4.1 for details on how labels are declared. The destination of a `GOTO` must be within the currently executing function or routine.

`FINISH` only remains in BCPL for historical reasons. It is equivalent to the call `stop(0, 0)` which causes the current program to stop execution. See the description of `stop(code, res)` page 48.

2.3.7 Compound Commands

It is often useful to be able to execute commands in a sequence, and this can be done by writing the commands one after another, separated by semicolons and enclosed in section brackets. The syntax is as follows:

$$\{\ C_1\ ;\ldots;\ C_m\ \}$$

where C_1 to C_m are commands.

Any semicolon ocurring at the end of a line may be omitted. For this rule to work, infixed expression operators may never start a line (see Section 2.2.9).

2.3.8 Blocks

A block is similar to a compound command but may start with some declarations. The syntax is as follows:

$$\{\ D_1\ ;\ldots;\ D_n;\ C_1\ ;\ldots;\ C_m\ \}$$

where D_1 to D_n are delarations and C_1 to C_m are commands. The declarations are executed in sequence to initialise any variables declared. A name may be used on the right hand side of its own and succeeding declarations and the commands (the body) of the block.

2.4 Declarations

Each name used in BCPL program must in the scope of its declaration. The scope of names declared at the outermost level of a program include the right hand side of its own declaration and all the remaining declarations in the section. The scope of names declared at the head of a block include the right hand side of its own declaration, the succeeding declarations and the body of the block. Such declarations are introduced by the keywords `MANIFEST`, `STATIC`, `GLOBAL` and `LET`. A name is also declared when it occurs as the control variable of a for loop. The scope of such a name is the body of the for loop.

2.4.1 Labels

The only other way to declare a name is as a label of the form N`:`. This may prefix a command or occur just before the closing section bracket of a compound command or block. The scope of a label is the body of the block or compound command in which it was declared.

2.4.2 Manifest Declarations

A `MANIFEST` declaration has the following form:

`MANIFEST` { N_1`=`K_1 `; . . . ;` N_n`=`K_n }

where N_1`, . . ,`N_n are names (see Section 2.2.1) and K_1`, . . ,`K_n are manifest constant expressions (see Section 2.2.10). Each name is declared to have the constant value specified by the corresponding manifest expression. If a value specification (`=`K_i) is omitted, the a value one larger than the previously defined manifest constant is implied, and if `=`K_1 is omitted, then `=0` is assumed. Thus, the declaration:

```
MANIFEST { A; B; C=10; D; E=C+100 }
```

declares `A`, `B`, `C`, `D` and `E` to have manifest values `0`, `1`, `10`, `11` and `110`, respectively.

2.4.3 Global Declarations

The global vector is a permanently allocated region of store that may be directly accessed by any (separately compiled) section of a program (see Section 2.5. It provides the main mechanism for linking together separately compiled sections. A GLOBAL declaration allows a names to be explicitly associated with elements of the global vector. The syntax is as follows:

$$\texttt{GLOBAL } \{\ N_1\texttt{:}K_1\ \texttt{;} \ldots \texttt{;}\ N_n\texttt{:}K_n\ \}$$

where $N_1, \ldots, N_n$ are names (see Section 2.2.1) and $K_1, \ldots, K_n$ are manifest constant expressions (see Section 2.2.10).

Each constant specifies which global vector element is associated with each variable.

If a global number ($\texttt{:}K_i$) is omitted, the next global variable element is implied. If $\texttt{:}K_1$ is omitted, then `:0` is assumed. Thus, the declaration:

```
GLOBAL { a; b:200; c; d:251 }
```

declares the variables `a`, `b`, `c` and `d` occupy positions 0, 200, 201 and 251 of the global vector, respectively.

2.4.4 Static Declarations

A `STATIC` declaration has the following form:

$$\texttt{STATIC } \{\ N_1\texttt{=}K_1\ \texttt{;} \ldots \texttt{;}\ N_n\texttt{=}K_n\ \}$$

where $N_1, \ldots, N_n$ are names (see Section 2.2.1) and $K_1, \ldots, K_n$ are manifest constant expressions (see Section 2.2.10). Each name is declared to be a statically allocated variable initialised to the corresponding manifest expression. If a value specification ($\texttt{=}K_i$) is omitted, the a value one larger than the previously defined manifest constant is implied, and if $\texttt{=}K_1$ is omitted, then `=0` is assumed. Thus, the declaration:

```
STATIC { A; B; C=10; D; E=C+100 }
```

declares `A`, `B`, `C`, `D` and `E` to be static variables having initial values `0`, `1`, `10`, `11` and `110`, respectively.

2.4.5 LET Declarations

LET declarations are used to declare local variables, vectors, functions and routines. The textual scope of names declared in a LET declaration is the right hand side of its own declaration (to allow the definition of recursive procedures), and subsequent declarations and the commands.

Local variable, vector and procedure declarations can be combined using the word AND. The only effect of this is to extend the scope of names declared forward to the word LET, thus allowing the declaration of mutually recursive procedures. AND serves no useful purpose for local variable and vector declarations.

2.4.6 Local Variable Declarations

A local variable declaration has the following form:

$$\texttt{LET}\ N_1\ \texttt{,...,}\ N_n\ \texttt{=}\ E_1\ \texttt{,...,}\ E_n$$

where N_1,..,N_n are names (see Section 2.2.1) and E_1,..,E_n are expressions. Each name, N_i, is allocated space in the current stack frame and is initialized with the value of E_i. Such variables are called dynamic variables since they are allocated when the declaration is executed and cease to exist when control leaves their scope. The variables N_1,..,N_n are allocated consecutive locations in the stack and so, for instance, the variable N_i may be accessed by the expression $(@N_1)!(\texttt{i} - \texttt{1})$. This feature is a recent addition to the language.

The query expression (?) should be used on the right hand side when a variable does not need an initial value.

2.4.7 Local Vector Declarations

$$\texttt{LET}\ N\ \texttt{=}\ \texttt{VEC}\ K$$

where N is a name and K is a manifest constant expression. A location is allocated for N and initialized to a vector whose lower bound is 0 and whose upper bound is K. The variable N and the vector elements ($N!0$ to $N!K$) reside in the runtime stack and only continue to exist while control remains within the scope of the declaration.

2.4.8 Procedure Declarations

A procedure declaration has the following form:

LET N (N_1 ,..., N_n) = E
LET N (N_1 ,..., N_n) BE C

where N is the name of the function or routine being declared, N_1,..,N_n are its formal parameters. A function is defined using = and returns E as result. A routine is defined using BE and executes the command C withou returning a result.

Some example declarations are as follows:

```
LET wrpn(n) BE { IF n>9 DO wrpn(n/10)
                 wrch(n REM 10 + '0')
               }
LET gray(n) = n NEQV n>>1
LET next() = VALOF { c := c-1
                     RESULTIS !c
                   }
```

If a procedure is declared in the scope of a global variable with the same name then the global variable is given an initial value representing the procedure (see section 2.5).

A procedure defined using equals (=) it is called a function and yields a result, while a procedure defines by BE is called a routine and does not. If a function is invoked as a routine its result in thrown away, and if a routine is invoked as a function its result is undefined. Functions and routines are otherwise similar. See section 2.2.3 for information about the syntax of to function and routine calls.

The arguments of a procedure behave like named elements of a dynamic vector and so exist only for the lifetime of the procedure call. This vector has as many elements as there are formal parameters and they receive their initial values from the actual parameters at the moment of call. Procedures are variadic; that is, the number of actual parameters need not equal the number of formals. If there are too few actual parameters then the missing higher numbered ones are left uninitialized, and if there are too many actual parameters, the extra ones are evaluated but their values discarded. Notice that the i^{th} argument can be accessed by the expression (@v)!i, where v is the first argument. The scope of the formal parameters is the body of the procedure.

Procedure calls are cheap in both space and execution time, with a typical space overhead of three words of stack per call plus one word for each formal parameter. In the Cintcode implementation, the execution overhead is typically just one executed insruction for the call and one for the return.

There are two important restrictions concerning procedures. One is that a `GOTO` command cannot make a jump to a label not declared within the current procedure, although such non local jumps can be made using the library procedures `level` and `longjump`, described on page 43. The other is that dynamic free variables are not permitted.

2.4.9 Dynamic Free Variables

Free variables of a procedure are those that are used but not declared in the procedure, and they are restricted to be either manifest constants, static variables, global variables, procedures or labels. This implies that they are not permitted to be dynamic variables (ie local variables of another procedure). There are several reasons for this restriction, including the need to be able to represent a procedure in a single word, the ability to provide a safe separate compilation facility with the related ability to assign procedures to variables. It also allows the procedure calling to be efficient. Programmers used to languages such as Algol or Pascal will find that they need to change their programming style somewhat; however, most experienced BCPL users agree that the restriction is well worthwhile. One should note that C adopted the same restriction, although in that language it is imposed by the simple expedient of insisting that all procedures are declared at the outermost level, thus making dynamic free variables syntactically impossible.

A style of programming that is often be used to avoid the dynamic free variable restriction is exemplified below.

```
GLOBAL { var:200 }

LET f1(...) BE
{ LET oldvar = var    // Save the current value of var
  var := ...          // Use var during the call of f1
  ...
  f2(...)             // var may be used in f2
  ...
  IF ... DO f1(...)   // f1 may be called recursively
  var := oldvar       // restore the original value of var
}

AND f2(...) BE        // f2 uses var as a free variable
{  ... var ...  }
```

2.5 Separate Compilation

Large BCPL programs can be split up into sections that can be compiled separately. When loaded into memory they can communicate with each other using a special area of store called the *Global Vector*. This mechanism is simple and machine independent and was put into the language since linkage editors at the time were so primitive and machine dependent.

Variables residing in the global vector are declared by GLOBAL declarations (see section 2.4.3). Such variables can be shared between separately compiled sections. This mechanism is similar to the used of BLANK COMMON in Fortran, however there is an additional simple rule to permit access to procedures declared in different sections.

If the definition of a function or routine occurs within the scope of a global declaration for the same name, it provides the initial value for the corresponding global variable. Initialization of such global variables takes place at load time.

The three files shown in Table 2.1 form a simple example of how separate compilation can be organised.

File `demohdr`	File `demolib.b`	File `demomain.b`

`GET "libhdr"`
 	
`GLOBAL { f:200 }`
	`GET "demohdr"`
	
	`LET f(...) = VALOF`
	`{ ...`
	`}`
		`GET "demohdr"`
		
		`LET start() BE`
		`{ ...`
		`  f(...)`
		`}`

Table 2.1 - Separate compilation example

When these sections are loaded, global 200 is initialized to the entry point of function `f` defined in `demolib.b` and so is can be called from the function `start` defined in `demomain.b`.

The header file, `libhdr`, contains the global declarations of all the resident library functions and routines making all these accessible to any section that started with: `GET "libhdr"`. The library is described in the next chapter. Global variable 1 is called `start` and is, by convention, the first function to be called when a program is run.

Automatic global initialisation also occurs if a label declared by colon (`:`) occurs in the scope of a global of the same name.

Although the global vector mechanism has disadvantages, particularly in the organisation of library packages, there are some compensating benefits arising

from its extreme simplicity. One is that the output of the compiler is available directly for execution without the need for a link editing step. Sections may also be loaded and unloaded dynamically during the execution of a program using the library functions `loadseg` and `unloadseq`, and so arbitrary overlaying schemes can be organised easily. An example of where this is used is in the implementation of the Command Language Interpreter described in Chapter 4. The global vector also allows for a simple but effective interactive debugging system without the need for compiler constructed symbol tables. Again, this was devised when machines were small and disc space was very limited; however, some of its advantages are still relevant today.

Chapter 3

The Library

This chapter describes the resident library functions, routines and manifest constants that are declared in the standard library header file `libhdr` and defined in either `SYSLIB` or `sys/BLIB.b`.

3.1 The Standard Header File `libhdr`

The file `libhdr` contains the global and manifest declarations of all the procedures, variables and constants belonging to the standard resident library. Global variables 0 to 199 are reserved for use by the system. Of these, globals 0 to 99 mainly belong to BLIB and SYSLIB, while tose between 133 and 149 belong to the Command Language Interpreter described in Chapter 4. The first global available to the user is numbered 200. For convenience this is declared as the manifest constant `ug`.

The size of the global vector is held in `globsize` (global 0) and, by convention, the global `result2` is used by some functions to return a second result. Global variable 1 is called `start` and is special since it must be defined by the user because it is the first function to be called when a program is run. Global 4 (called `clihook` is also special since it is useful when debugging programs, see page 47.

3.1.1 Architecture Constants

This constant `B2Wsh` holds the shift required to convert a BCPL pointer into a byte address. On 32 bit machines it is set to 2 while on some 64 bit implementations it is set to 3. The constant `bytesperword` is defined to be `1<<B2Wsh` and indicates how many bytes can be packed into a word. The constant `bitsperbyte` is set to 8 indicating the size of a byte. The constants `minint` and `maxint` hold

the largest negative and positive numbers that can be represented by a word in this implementation.

3.1.2 Other Manifest Constants

The constants `t_hunk`, `t_bhunk` and `t_end` are used in the representation of Cintcode Object Modules. Constants whose names start with `co_` are used in the implementation of the coroutine mechanism (see Section 3.3.11), and constants starting with `rtn_` denote offsets in the rootnode (see Section 3.3.1). The constants `InitObj`(=0) and `CloseObj`(=1) are the positions in the methods vector of the routines to initialise and close an object. See `mkobj` described on page 46.

3.2 SYSLIB

This module contains the definitions of the functions `sys`, `chgco` and `muldiv`. These functions are hand written in Cintcode because they need to execute the Cintcode instructions (`SYS`, `CHGCO` and `MDIV`) that cannot be generated by the BCPL compiler. The instruction `SYS` provides an interface with the host operating system, `CHGCO` is used in the implementation of coroutines (see Section 3.3.11), and `MDIV` performs the operation required by `muldiv`(see Section 3.3.13).

To understand the purpose of `sys` it is necessary to known some how the Cintcode System in implemented. It is mainly implemented in C and can be found in the files `sys/cintmain.c`, `sys/cinterp.c`, `sys/kblib.c` and `sys/nrastlib.c`. There is also a header file `sys/cinterp.h` that contains architecture specific declarations. The file `sys/cintmain.c` contains the main program and includes the definition `dosys` which provide access to I/O functions and other operating system primitives. The file `sys/cinterp.c` contains the C implementation of the Cintcode interpreter and there is usually a more efficient interpreter. See, for instance, `sys/LINUX/cintasm.s`. `cintasm` is faster than the C version because it has fewer built-in debugging aids and because it is carefully hand written in assembly language taking advantage of knowledge not available to a C compiler. The BCPL function `sys` provides an interface between BCPL and `dosys`, and can also control which version of the interpreter is used.

3.2.1 sys

The `sys` function is defined by hand in SYSLIB and just invokes the `SYS` Cintcode instruction. When `SYS` is encountered by the interpreter, it normally just calls `dosys` passing the BCPL P and G pointers as arguments. But when `sys(0,code)`

is executed, the interpreter saves the Cintcode registers in a vector and returns with the result `code` to the (C) program that called this invocation of the interpreter. This is normally used to exit from the Cintcode system, but can also be used to return from recursive invocations of the interpreter (see `sys(1,regs)` below).

A typical call of `sys` is as follows:

res `:= sys(`*op*`,...)`

The action performed by this call depends on *op*. Some actions concern the management of the interpreter, some are concerned with input and output, while others provide access to functions implemented in C.

3.2.2 Interpreter Management

The Cintcode System normally has two resident interpreters. One is called `cinterp` and is implemented in C and the other is called `cintasm` and is normally implemented by hand in assembly language. The assembly language version runs faster but provides no way of counting instruction executions or profiling program execution. It is possible to select dynamically which interpreter is running by setting a value in the Cintcode `count` register. A positive value causes `cinterp` to run. Each time `cinterp` executes a Cintcode instruction it decrements `count` and returns with an error code of `3` when `count` reaches zero. If the value in `count` is `-1`, `cintasm` is invoked and runs without changing `count`. With some debugging versions of `cintasm`, setting `count` to `-2` causes `cintasm` to execute just one instruction and then return with error code `10`. This feature is provided to assist the debugging of a new version of `cintasm`. The Cincode `count` register can be set by means of the `sys(-1,...)` call described below.

A second version of the BCPL Cintcode system called `rasterp`. It uses just one interpreter implemented in C, and can be called upon to generate raster image data. For more information, see the description of `sys(27,...)` on page 35 and the `raster` and `rast2ps` commands on page 63.

oldcount `:= sys(-1,` *newcount*`)`

Under normal conditions when the interpreter is running under the control of code in `BOOT`, this sets the Cintcode count register to *newcount* and returns the previous value. If the count register is now less than `0`, interpretation continues using the fast interpreter (`cintasm`), but if it is greater than `0`, interpretation continues using the slow interpreter (`cinterp`). The code to make this selection can be found in the execution loop occuring at the end of the function `start` defined in `sys/BOOT.b`.

For some (debugging) versions of `cintasm` a count of `-2` causes the (fast) interpreter to execute just one Cintcode instruction before returning with an error code of `10`.

The `sys(-1,...)` function is used by the CLI command `interpreter` to select which interpreter is to be used (see Section 4.2). It is also used by the `instrcount` function defined in `BLIB.b` (see page 48).

`sys(0,` *code*`)`

This will cause a return from the the interpreter yielding *code* as the return code. A *code* of zero denotes successful completion and, if invoked at the outermost level, causes the BCPL Cintcode System to terminate.

res `:= sys(1,` *regs*`)`

This function enters the Cintcode interpreter recursively with the Cintcode registers set to values specified in the vector *regs*. The elements of *regs* are as follows:

regs`!0`	A register	– work register
regs`!1`	B register	– work register
regs`!2`	C register	– work register
regs`!3`	P register	– the stack frame pointer
regs`!4`	G register	– the base of the global vector
regs`!5`	ST register	– the status register (unused)
regs`!6`	PC register	– the program counter
regs`!7`	Count register	– see below

When `cinterp` is active, the count register is decremented every time an instruction is interpreted. When the count goes negative the interpreter saves the registers and returns with a result (=3) to indicate what has happened. If the count register is positive it indicates how many Cintcode instructions should be executed before the interpreter returns. A count of `-1` is treated as infinity and causes the fast interpreter `cintasm` to be used.

Either interpreter returns when a fault, such as division by zero, occurs or when a call of `sys(0,...)` or `sys(-1,...)` is made. When returning, the current state of the Cintcode registers is saved back into *regs*. The returned result is either the second argument of `sys(0,...)` or one of the builtin return codes in the following table:

-1	Re-enter the interpreter with a new value in the the count register
0	Normal successful completion (by convention)
1	Non existant Cintcode instruction
2	BRK instruction encountered
3	Count has reached zero
4	PC set to a negative value
5	Division by zero
10	Single step interrupt from the fast interpreter (debugging)

`sys(2)` – Turn on tracing of the Cintcode instruction execution.
`sys(3)` – Turn off tracing.
`sys(4)` – Clear the tally vector and start tallying.
`sys(5)` – Stop tallying.

When running under `cinterp`, the calls `sys(2)` and `sys(3)` provide a primitive trace facility to aid debugging the `cinterp` itself and are not normally of use to ordinary users. When running under `cinterp`, the calls `sys(4)` and `sys(5)` provide the profiling facility that was used to obtain execution statistics. It uses the tally vector to hold frequency counts of Cintcode instructions executed. When tallying is enabled, the i^{th} element of the tally vector is incremented every time the instruction at location i of the Cintcode memory is executed. The upper bound (normally 80000) of the tally vector is implementation dependent but is stored in the zeroth element of the vector. The location of the tally vector can be found by evaluating the expression `rootnode!rtn_tallyv`. Note this profiling facility is only available when running under `cinterp`. Statistics of the execution of a program can be gathered and analysed using the CLI command `stats` (see Section 4.2).

3.2.3 Primitive I/O Operations

ch `:= sys(10)`

Return the next character from the keyboard. The character is echoed to standard output (normally the screen).

`sys(11,` *ch*`)`

Send character *ch* to the standard output (normally the screen). the character linefeed is transmitted as carriage return followed by linefeed.

n `:= sys(12,` *fp*`,` *buf*`,` *len*`)`

Read upto *len* bytes from the file specified by the file pointer *fp* into the byte

buffer *buf*. The file pointer must have been created by a call of `sys(14,...)`. The number of bytes actually read is returned as the result.

n := `sys(13,` *fp*`,` *buf*`,` *len*`)`

Write *len* bytes to the file specified by the file pointer *fp* from the byte buffer *buf*. The file pointer must have been created by a call of `sys(15,...)`. The result is the number of bytes transferred, or zero if there was an error.

fp := `sys(14,` *name*`)`

This opens for reading the file whose name is given by the string *name*. It returns `0` if the file cannot be opened, otherwise it returns the file pointer for the opened file. See page 43 for information about the treatment of file names.

fp := `sys(15,` *name*`)`

This opens for writing the file whose name is given by the string *name*. It returns `0` if the file cannot be opened, otherwise it returns the file pointer for the opened file. See page 43 for information about the treatment of file names.

res := `sys(16,` *fp*`)`

This closes the file whose file pointer is `fp`. It return 0 if successful.

res := `sys(17,` ***name***`)`

This deletes the file whose name is given by `name`. See page 43 for information about the treatment of file names.

res := `sys(18,` *old*`,` *new*`)`

This renames file *old* to *new*. It return 0 if successful.

res := `sys(21,` *upb*`)`

This allocates a vector whose lower bound is 0 and whose upper bound is *upb*. It return zero if the request cannot be satified.

`sys(22,` *ptr*`)`

If *ptr* is zero it does nothing, otherwise it returns the space pointed to by *ptr* that must have previously been allocated by `sys(21,...)`.

res := `sys(23,` *name*`)`

This loads a Cintcode module from file *name*. It return a pointer to the loaded module if sucessful. Otherwise it returns zero.

res := `sys(24,` *seg*`)`

This initializes the global variables define in the loaded module pointed to by *seg*. It returns zero is there is an error.

res := sys(25, *seg*)

This unloads the the loaded module given by *seg*. If *seg* is zero it does nothing.

res := sys(26, *a*, *b*, *c*)

This invoke the C implementation of `muldiv`. It returns the result of dividing *c* into the double length product of *a* and *b*. It sets `result2` to the remainder.

res := sys(27, *n*, *arg*)

This invokes the `setraster` primitive used in the rastering version of the Cintcode interpreter (`rasterp`). It is used to control the collection of data for time-memory raster images. If *n*=3, it returns `0` if rastering is available and `-1` otherwise. If *n*=2, the memory granularity is set to *arg* bytes per pixel. The default is 12. If *n*=1, the number of Cintcode instructions per raster line is set to *arg*. The default is 1000.

If *n* is zero and *arg* is non-zero then rastering is activated to send its output to file *name* (the rastering data file). Raster information is collected for the duration of the next CLI command. If *n* and *arg* are both zero the rastering data file is closed.

The raster data file is an ASCII file that encodes the raster lines using run length encoding. Typical output is as follows:

`K1000 S12`	1000 instruction per raster line, 12 bytes per pixel
`W10B3W1345B1N`	10 white, 3 black, 1345 white, 1 black, newline
`W13B3W12B2N`	etc
`...`	

See the CLI commands `raster` and `rast2ps` on page 63 for more information on how to use the rastering facility.

res := sys(28)

This returns `TRUE` if the program is being interrupted by the user. On many systems this is not implemented and just returns `FALSE`.

res := sys(30)

This returns the CPU time in milliseconds since the Cintcode system was entered.

res := sys(31, *name*)

This returns time of last modification of the file given by *name*.

res `:= sys(32,` *prefix*`)`

This is primarily a function for the Windows CE version of the BCPL Cintcode System for which there is no current working directory mechanism. It sets the prefix that is prepended to all future relative file names. See Sections 3.3.5 and the CLI `prefix` command described on page 63.

str `:= sys(33)`

This returns the current prefix string. See `sys(32,...)` above.

res `:= sys(34,...)`

This is currently only useful on the Windows CE version of the BCPL Cintcode system. It performs an operation on the graphics window. The graphics window is a fixed size array of 8-bit pixels which can be written to and whose visibility can be switched on and off.

res `:= sys(35)`

This returns `TRUE` if a character is ready to be read from the keyboard. This function is currently only available in the Windows CE implementation.

3.3 BLIB

`BLIB` contains the main part of the standard library. It is implemented in BCPL and its source code can be found in `sys/BLIB.b`.

3.3.1 Initialization

When the Cintcode System is started, a region of store is allocated for the Cintcode memory. This is where Cintode stacks, global vectors, program code and system data is placed. Within it there is a vector called the *rootnode* that allows running programs to locate components of the system data structure. The global variable `rootnode` holds a pointer to the rootnode and there are manifest constants (define in `libhdr`) to ease access to its various elements. For instance, the pointer to the start of the memory block chain can be obtained by evaluating `rootnode!rtn_blklist`. Ten elements are defined in the rootnode as shown

below.

Expression	Value
rootnode!rtn_membase	Pointer to the start of the Cintcode memory.
rootnode!rtn_memsize	The size of the Cintcode memory in words.
rootnode!rtn_blklist	The start of the chain of memory blocks.
rootnode!rtn_tallyv	The tally vector.
rootnode!rtn_syslib	The SYSLIB code segment.
rootnode!rtn_blib	The BLIB code segment.
rootnode!rtn_boot	The BOOT code segment.
rootnode!rtn_cli	The CLI code segment.
rootnode!rtn_keyboard	The stream control block for the keyboard.
rootnode!rtn_screen	The stream control block for the screen.

v := getvec(*upb*)
freevec(*v*)

Allocation and release of space is performed by getvec and freevec, which use a first fit algorithm based on a list of blocks chained together in memory order. Word zero of each block in the chain contains a flag in its least significant bit indicating whether the block is allocated or free. The rest of the word is an even number giving the size of the block in words. A pointer to the first block in the chain is held in the rootnode which is used to hold system wide information.

getvec allocates a vector with upper bound *upb* from the first large enough free block on the block list. If no such block exists it returns zero. A vector previously allocated by getvec can be freed by the above call of freevec. Coalescing of adjacent free blocks is performed by getvec.

ch := sardch()
sawrch(*ch*)
sawritef(*format*, *a*, *b*, ...)

These functions provide standalone input and output designed primarily as an aid for debugging the system before the full stream based I/O mechanism is available.

The function sardch returns the next character from the keyboard as soon as it is available, echoing the character to the screen. It is implemented by means of sys(10), described above.

The call sawrch(*ch*) outputs the character *ch* to the screen. It is implemented by means of sys(11,*ch*), described above. The function sawritef is similar to writef but performs its output using sawrch.

3.3.2 Stream Input/Output

BCPL uses streams as a convenient method of obtaining device independent input and output. All the information needed to process a stream is held in a vector called a stream control block (SCB) whose structure is shown in figure 3.1.

The elements `pos` and `end` hold positions within the byte buffer, `file` holds a file pointer for file streams or `-1` for streams connected to the console. The element `id` indicates whether the stream is for input or output and work is private work space for the action procedures `rdfn`, `wrfn` which are called, repectively, when the byte buffer becomes empty on reading or full on output. The procedure `endfn` is called to close the stream.

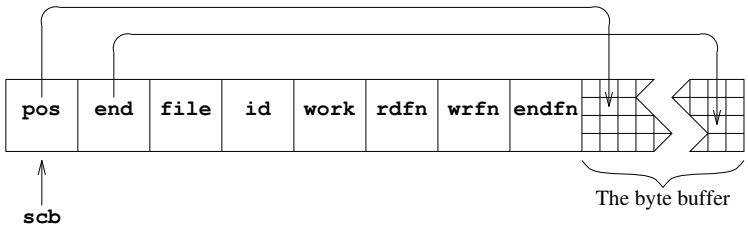


Figure 3.1: Stream control block structure

Input is read from the currently selected input stream whose SCB is held in the global variable `cis`. For an input stream, the SCB element `pos` holds the byte offset of the position of the next character to be read, and `end` holds the offset of the position just past the last character currently available in the buffer. Characters are read using `rdch` whose definition is given in figure 3.2. If a character is available in the byte buffer this is returned after incrementing `pos`, otherwise the element `rdfn` is called to replenish the buffer and return its first character.

```
LET rdch() = VALOF
{ LET pos = cis!scb_pos
  IF pos < cis!scb_end DO { cis!scb_pos := pos+1
                            RESULTIS cis%pos
                          }
  RESULTIS (cis!scb_rdfn)(cis)
}
```

Figure 3.2: The definition of `rdch`

The buffer always contains the previously read character whenever possible. This is to allow for a clean and simple implementation of `unrdch` whose purpose is to step input back by one character position. Its definition if given in figure 3.3.

```
LET unrdch() = VALOF
{ LET pos = cis!scb_pos
  IF pos<=scb_bufstart RESULTIS FALSE // Cannot UNRDCH past origin.
  cis!scb_pos := pos-1
  RESULTIS TRUE
}
```

Figure 3.3: The definition of `unrdch`

Output is sent to the currently selected output stream whose SCB is held in the global variable `cos`. For an output stream, the SCB element `pos` holds the byte offset of the position of the next character to be written, and `end` holds the offset of the position just past the end of the buffer. Characters are written using the function `wrch` whose definition is given in figure 3.4. The character `ch` is copied into the byte buffer and `pos` incremented. If the buffer is now full, it is emptied and reset by calling the element `wrfn`. If writing fails it return `FALSE`, causing `wrch` to abort.

```
LET wrch(ch) BE
{ LET pos = cos!scb_pos
  cos%pos := ch
  cos!scb_pos := pos+1
  IF pos>=cos!scb_end UNLESS (cos!scb_wrfn)(cos) DO abort(189)
}
```

Figure 3.4: The definition of `wrch`

The SCB for the screen has a buffer size of one to ensure that each call of wrch transmits its character to the screen. The size of the other stream buffers is 2048 bytes.

3.3.3 Input Functions

scb `:= findinput(`*name*`)`
scb `:= findinput(`*name*`,` *pathname*`)`
`selectinput(`*scb*`)`
ch `:= rdch()`
flag `:= unrdch()`
scb `:= input()`
`endread()`
n `:= readn()`

The call `findinput(`*name*`)` opens an input stream. If *name* is the string `"*"` then it opens the standard input stream which is normally from the keyboard, otherwise *name* is taken to be a device or file name. If the stream cannot be opened the result is zero. See Section 3.3.5 for information about the treatment of filenames.

The call `findinput(`*name*`, `*pathname*`)` opens an input stream. If *name* is the string `"*"` then input comes from the keyboard, otherwise *name* is taken to be a filename. If the stream cannot be opened the file directories specified by the shell variable *pathname* are searched. If the file is not found in any of these directories, the result is zero. The shell variable `BCPLPATH` is used by the Command Language Intepreter (see Chapter 4) when searching for commands, and by the BCPL compiler when processing `GET` directives.

The call `selectinput(`*scb*`)` selects *scb* as the currently selected input stream. It aborts (with code 186) if *scb* is not an input stream.

The call `rdch()` reads the next character from the currently selected input stream. If the stream is exhausted, it returns the special value `endstreamch`. Input from the keyboard is buffered until the ENTER (or RETURN) key is pressed to allow simple line editing in which the backspace key may be used to delete the most recent character typed.

The call `unrdch()` attempts to step the current input stream back by one character position. It returns `TRUE` if successful, and `FALSE` otherwise. A call of `unrdch` will always succeeds the first time after a call of `rdch`. It is useful in functions such as `readn` (described below) where single character lookahead is necessary.

The call `input()` returns the currently selected input stream.

The call `endread()` closes the currently selected input stream. If there is an error it aborts (with code 190).

The call `readn()` reads an optionally signed decimal integer from the currently selected input stream. Leading spaces, tabs and newlines are ignored. If the number is syntactically correct, `readn` returns its value with `result2` set to zero, otherwise it returns zero with `result2` set to `-1`. In either case, it uses `unrdch` to replace the terminating character.

3.3.4 Output Functions

scb `:= findoutput(`*name*`)`

This function opens an output stream specified by the device or file name *name*. If *name* is the string `"*"` then it opens the standard output stream which

is normally to the screen. If the stream cannot be opened, the result is zero. See Section 3.3.5 for information about the treatment of filenames.

`selectoutput(`*scb*`)`

This routine selects *scb* as the currently selected output stream. It aborts (with code 187) if `scb` is not an output stream.

`wrch(`*ch*`)`

This routine writes the character *ch* to the currently selected output stream. If output is to the screen, *ch* is transmitted immediately. It aborts (with code 189) if there is a write failure.

scb `:= output()`

This function returns the SCB of the currently selected output stream.

`endwrite()`

This routine closes the currently selected output stream. If there is an error it aborts with code 189 (trouble with writing) or code 191 (trouble with closing).

`newline()`
`newpage()`

These two routines simply output the newline character (`'\n'`) or the newpage (form-feed) character (`'\p'`), respectively, to the currently selected output stream.

`writed(`*n*`, `*d*`)`
`writeu(`*n*`, `*d*`)`
`writen(`*n*`)`

These routines output the integer *n* in decimal to the currently selected output stream. For `writed` and `writeu`, the output is padded with leading spaces to fill a field width of *d* characters. If `writen` is used or if *d* is too small, the number is written without padding. If `writeu` is used, *n* is regarded as an unsigned integer.

`writehex(`*n*`, `*d*`)`
`writeoct(`*n*`, `*d*`)`
`writebin(`*n*`, `*d*`)`

These routines output, repectively, the least significant *d* hexadecimal, octal or binary digits of the integer *n* to the currently selected output stream.

`writes(`*str*`)`
`writet(`*str*`, `*d*`)`

These routines output the string *str* to the currently selected output stream. If `writet` is used, trailing spaces are added to fill a field width of `d` characters.

`writef(`*format*`,`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)`

The first argument (*format*) is a string that is copied character by character to the currently selected output stream until a substitution item such as `%s` or `%i5` is encountered when a value (usually the next argument) is output in the specified format. The substitution items are given in table 3.5.

Item	Substitution
`%s`	Write the next argument as a string using `writes`.
`%t`*n*	Write the next argument as a left justified string in a field width of *n* characters using `writet`.
`%c`	Write the next argument as a character using `wrch`.
`%b`*n*	Write the next argument as a binary number in a field width of *n* characters using `writebin`.
`%o`*n*	Write the next argument as an octal number in a field width of *n* characters using `writeoct`.
`%x`*n*	Write the next argument as a hexadecimal number in a field width of n characters using `writehex`.
`%i`*n*	Write the next argument as a decimal number in a field width of *n* characters using `writed`.
`%n`	Write the next argument as a decimal number in its natural field width using `writen`.
`%u`*n*	Write the next argument as an unsigned decimal number in a field width of *n* characters using `writeu`.
`%$`	Skip over the next argument.
`%%`	Write the character `%`.

Figure 3.5: `writef` substitution items

When a field width (denoted by *n* in the table) is required, it is specified by a single character, with 0 to 9 being represented by the corresponding digit and 10 to 35 represented by the letters `A` to `Z`. Format characters are case insensitive but field width characters are not.

The implementation of `writef` (in `sys/BLIB.b`) is a good example of how a variadic function can be defined.

3.3.5 The Filing System

BCPL uses the filing system of the host machine and so such details as the maximum length of filenames or what characters they may contain are machine dependents. However, within a file name the characters slash (/) and backslash (\) are regarded as a file separators and are converted into the appropriate separator for the operating system being used. For Unix systems this is a slash, for MS-DOS, WINDOWS and OS/2 it is a backslash, and on the Apple Macintosh it is a colon. Thus, under MS-DOS, `findoutput` can be given a file name such as `"tmp/RASTER"` and it will be treated as if the name `"tmp\RASTER"` had been given. This somewhat ad hoc feature greatly improves portability between systems.

A file name prefix feature is available primarily for systems such as Windows CE where there is no concept of a current working directory. The system maintains a prefix that is prepended to any non absolute file name before it is passed to the operating system. A file name is absolute if it starts with a slash or backslash or, on Windows systems, if it starts with a letter followed by a colon. A separator is placed between the prefix and the given file name.

The current prefix can be inspected and changed using the calls: `sys(32,`*prefix*`)` and `sys(33)`, or the CLI command `prefix` described on page 63.

3.3.6 File Deletion and Renaming

flag `:= deletefile(`*name*`)`
flag `:= renamefile(`*oldname*`, `*newname*`)`

The call `deletefile(`*name*`)` deletes the file with the given name. It returns `TRUE` if the deletion was successful, and `FALSE` otherwise. The call `renamefile(`*oldname*`, `*newname*`)` renames the file *oldname* as file *newname*, deleting *newname* if necessary. Both *oldname* and *newname* are strings. The function returns `TRUE` if the renaming was successful, and `FALSE` otherwise.

3.3.7 Non Local Jumps

P `:= level()`
`longjump(`*P*`, `*L*`)`

The call `level()` returns the current stack frame pointer for use in a later call of `longjump`. The call `longjump(`P`, `L`)` causes execution to resume at label L in the body of a procedure that owns the stack frame given by P. Jumps to labels within the current procedure can be performed using the `GOTO` command, so `level` and `longjump` are only needed for non local jumps.

3.3.8 Command Arguments

This implementation of BCPL incorporates a command language interpreter which is described in Chapter 4. Most commands require argument and these are most easily read using the functions: `rditem`, `rdargs`, `findarg` and `str2numb`.

kind `:= rditem(`v`, `*upb*`)`

The function `rditem` reads a command argument from the currently selected input stream. After ignoring leading spaces and tab characters, it packs the item into the vector v whose upper bound is *upb* and returns an integer describing the kind of item read. Table 3.6 gives the kinds of item that can be read and corresponding item codes.

Example items	Kind of item	Item code
`;`		4
carriage return		3
`"from"` `"\ntwo words\n"`	Quoted string	2
`abc` `123-45*6`	Unquoted string	1
end-of-stream	Terminator	0
	An error	-1

Figure 3.6: `rditem` results

It is possible to include newline characters within quoted strings using the escape sequence `*n`.

res `:= rdargs(`*keys*`, `*argv*`, `*upb*`)`

The first argument (*keys*) is a string specifying a list of argument keywords with possible qualifiers. The second and third arguments provide a vector (*argv*) with a given upper bound (*upb*) in which the decoded arguments are to be placed.

If `rdargs` is successful, it returns the number of words used in *argv* to represent the decoded command arguments, and, on failure, it returns zero.

Command arguments are read from the currently selected input stream using a decoding mechanism that permits both positional and keyed arguments to be freely mixed. A typical use of `rdargs` occurs in the source of the `input` command as follows:

```
UNLESS rdargs("FROM/A,TO/K,N/S", argv, 50) DO
{ writef "Bad arguments for INPUT\n"
  ...
}
```

In this example, there are three possible arguments and their values will be placed in the first three elements of `argv`. The first argument has keyword `FROM` and must receive a value because of the qualifier `/A`. The second has keyword `TO` and its qualifier `/K` insists that, if the argument is given, it must be introduced by its keyword. The third argument has the qualifier `/S` indicating that it is a switch that can be turned on by the presence of its keyword. If an argument is supplied, the corresponding element of `argv` will be set to `-1`, if it is a switch argument, otherwise it will be set to a string containing the characters of the argument value. The elements of `argv` corresponding to unset arguments are cleared. Table 3.7 shows the values in placed in `argv` and the result when the call:

`rdargs("FROM/A,TO=AS/K,N/S", argv, 50)`

is given various argument strings. This example illustrates that keyword synonyms can be defined using `=` within the key string. Positional arguments are those not introduced by keywords. When one is encountered, it becomes the value of the lowest numbered unset non-switch argument.

Arguments	argv!0	argv!1	argv!2	Result
abc TO xyz	"abc"	"xyz"	0	>0
to xyz from abc	"abc"	"xyz"	0	>0
as xyz abc n	"abc"	"xyz"	-1	>0
abc xyz	-	-	-	=0
"from" to "to"	"from"	"to"	0	>0

Figure 3.7: `rdargs("FROM/A,TO=AS/K,N/S", argv, 50)`

n `:= findarg(`*keys*`,` *item*`)`

The function `findarg` was primarily designed for use by `rdargs` but since it

is sometimes useful on its own, it is publicly available. Its first argument, *keys*, is a string of keys of the form used by `rdargs` and item is a string. If the result is positive, it is the argument number of the keyword that matches item, otherwise the result is `-1`.

n `:= str2numb(`*str*`)`

This function converts the string *str* into an integer. Characters other than `0` to `9` and `-` are ignored.

n `:= randno(`*upb*`)`

This function returns a random integer in the range `1` to `upb`. Its implementation is as follows:

```
STATIC { seed=12345 }

LET randno(upb) = VALOF
{ seed := seed*2147001325 + 715136305
  RETURN ABS(seed/3) REM upb + 1
}
```

oldseed `:= setseed(`*newseed*`)`

The current seed can be set to *newseed* by the call `setseed(`*newseed*`)`. This function returns the previous seed value.

obj `:= mkobj(`*upb*, *fns*, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*`)`

This function creates and initialises an object. It definition is as follows:

```
LET mkobj(upb, fns, a, b, c, d, e, f, g, h, i, j, k) = VALOF
{ LET obj = getvec(upb)
  UNLESS obj=0 DO
  { !obj := fns
    InitObj#(obj, @a) // Send the init message to the object
  }
  RETURN obj
}
```

As can be seen, it allocates a vector for the fields of the object, initialises its zeroth element to point to the methods vector and calls the initialisation method that is expected to be in element `InitObj` of *fns*. The result is a pointer to the initialised fields vector. If it fails, it returns zero. As can be seen the initialisation method receives a vector of up to 11 initialisation arguments.

3.3.9 Program Loading and Control

In this implementation, the BCPL compiler generates a file of hexadecimal numbers for the compiled code. For instance the compiled form of the `logout` command:

```
SECTION "logout"
GET "libhdr"
LET start() BE abort(0)
```

is

```
000003E8 0000000C
0000000C 0000FDDF 474F4C07 2054554F 0000DFDF
61747307 20207472 7B1F2310 00000000 00000001
0000001C 0000001F
```

The first two words indicate the presence of a "hunk" of code of size 12(`000000C`) words which then follow. The first word of the hunk (`000000C`), is again the length. The next two words (`0000FDDF` and `474F4C07`) contain the `SECTION` name `"logout"`. These are followed by the two words `0000DFDF` and `61747307` which identify the procedure name `"start"`. The body of `start` is compiled into one word (`7BF1F2310`) which correspond to the Cintcode instruction:

`L0`	Load A with 0
`K3G 31`	Call the function in global 31, incrementing the stack by 3
`RTN`	

The remaining 4 words contains global initialisation data indicating that global 1 is to be set to the entry point at position 28 (`0000001C`) relative to the start of the hunk, and that the highest referenced global number is 31 (`0000001F`).

code `:= start(`*a1*`, `*a2*`, `*a3*`, `*a4*`)`

This function is, by convention, the main procedure of a program. If it is called from the command language interpreter (see section 4), its first argument is zero and its result should be the command completion code; however, if it is the main procedure of a module run by `callseg`, defined below, then it can take up to 4 arguments and its result is up to the user. By convention, a command completion code of zero indicates successful completion and larger numbers indicate errors of ever greater severity

`clihook()`

This procedure is defined in BLIB and simply calls `start`. Its purpose is to

assist debugging by providing a place to set a breakpoint in the command language interpreter (CLI) just before a command in entered. Its is also permissible for the user to override the standard definition of clihook with a private version.

`stop(`*code*`)`

This function is provided to stop the execution of the current command running under control of the CLI. Its argument *code* is the command completion code.

count`:= instrcount(`*fn*`,`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)`

This procedure returns the number of Cintcode instruction executed when evaluating the call: *fn*`(`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)`.

Counting starts from the first instruction of the body of *fn* and ends when its final `RTN` instruction is executed. Thus when `f` was defined by `LET f(x) = 2*x+1`, the call `instrcount(f, 10)` returns 4 since its body executes the four instructions: `L2; MUL; A1; RTN`. The value returned by *fn*`(`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)` is saved by `instrcount` in the global variable `result2`.

`abort(`*code*`)`

This procedure causes an exit from the current activation of the interpreter, returning *code* as the fault code. If *code* is zero execution leaves the Cintcode system altogether, if *code* is `-1` execution resumes using the faster version of the interpreter (`cintasm`). If *code* is positive, under normal conditions, the interactive debugger is entered.

flag `:= intflag()`

This function provides a machine dependent test to determine whether the user is asking to interrupt the normal execution of a program. On the Apple Macintosh flag will be set to `TRUE` only if the COMMAND, OPTION and SHIFT keys are simultaneously pressed.

segl `:= loadseg(`*name*`)`

This function loads the compiled program into memory from the specified file name. It return the list of loaded program modules if loading was successful and zero otherwise. It does not initialise the global variables defined in the program.

res `:= globin(`*segl*`)`

This function initialises the global variables defined in the list of program modules given by its argument *segl*. It returns zero if the global vector was too small, otherwise it returns *segl*.

`unloadseg(`*segl*`)`

This routine unloads the list of loaded program modules given by *segl*.

res `:= callseg(`*name*`,`*a1*`,`*a2*`,`*a3*`,`*a4*`)`

This function loads the compiled program from the file *name*, initialises its global variables and calls `start` with the four arguments *a1*, . . . , *a4*. It returns the result of this call, after unloading the program.

3.3.10 Character Handling

ch `:= capitalch(`*ch*`)`

This function converts lowercase letters to uppercase, leaving other characters unchanged.

res `:= compch(`*ch1*`,` *ch2*`)`

This function compares two characters ignoring case. It yields `-1` (`+1`) if *ch1* is earlier (later) in the collating sequence than *ch2*, and `0` if they are equal.

res `:= compstring(`*s1*`,` *s2*`)`

This function compares two strings ignoring case. It yields `-1` (`+1`) if *s1* is earlier (later) in the collating sequence than *s2*, and `0` if they the strings are equal.

3.3.11 Coroutines

BCPL uses a stack to hold function arguments, local variables and anonymous results, and it uses the global vector and static variables to hold non-local quanities. It is sometimes convenient to have separate runtime stacks so that different parts of the program can run in pseudo parallelism. The coroutine mechanism provides this facility.

In this implementation, they have distinct stacks but share the same global vector, and it is natural to represent a coroutine by a pointer to its stack. At the base of each stack there are six words of system information as shown in figure 3.8.

The resumption point is P pointer belonging to the procedure that caused the suspension of the coroutine. It becomes the value of the P pointer when the coroutine next resumes execution. The parent link points to the coroutine that called this one, or is zero if the coroutine not active. The outermost coroutine

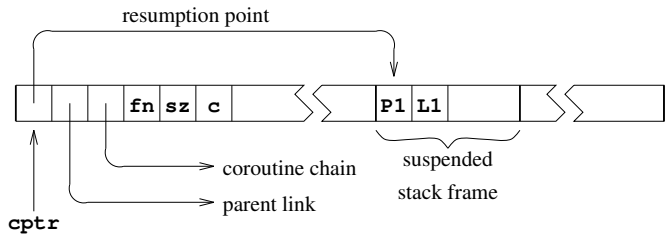


Figure 3.8: A coroutine stack

(or *root coroutine*) is marked by the special value `-1` in its parent link. As a debugging aid, all coroutines are chained together in a list held in the global `colist`. The values `fn` and `sz` hold the main function of the coroutine and its stack size, and `c` is a private variable used by the coroutine mechanism.

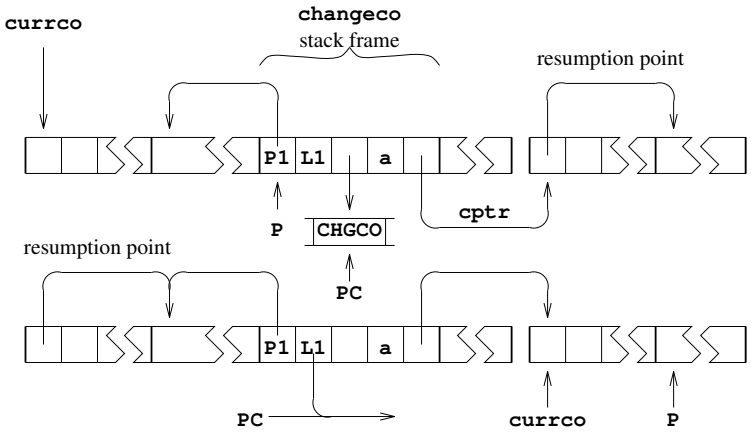


Figure 3.9: The effect of `changeco(a, cptr)`

At any time just one coroutine (the *current coroutine*) has control, and all the others are said to be suspended. The current coroutine is held in the global variable `currco`, and the P pointer points to a stack frame within its stack. Passing control from one coroutine to another involves saving the resumption point in the current coroutine, and setting new values for the program counter (PC), the P pointer and `currco`. This is done by `changeco(a,cptr)` as shown in figure 3.9. The function `changeco` is defined by hand in `SYSLIB` and its body consists of the single instruction `CHGCO` and as can be seen its effect is somewhat subtle. The only uses of `changeco` are in the definitions of `createco`, `callco`, `cowait` and `resumeco`, and its effect, in each case, is explained below. The only functions that can cause coroutine suspension are `callco`, `cowait` and `resumeco`.

res `:= callco(`*cptr*`,` *arg*`)`

This call suspends the current coroutine and transfers control to the coroutine

pointed to by *cptr*. It does this by resuming execution of the function that caused its suspension, which then immediately returns yielding *arg* as result. When `callco(`*cptr*`,`*arg*`)` next receives control it yields the result it is given.

res `:= cowait(`*arg*`)`

This call suspends the current coroutine and returns control to its parent by resuming execution of the function that caused its suspension, yielding *arg* as result. When `cowait(`*arg*`)` next receives control it yields the result it is given.

res `:= resumeco(`*cptr*`,` *arg*`)`

The effect of `resumeco` is almost identical to that of `callco`, differing only in the treatment of the parent. With `resumeco` the parent of the calling coroutine becomes the parent of the called coroutine, leaving the calling coroutine suspended and without a parent. Systematic use of `resumeco` reduces the number of coroutines having parents and hence allows greater freedom in organising the flow of control between coroutines.

cptr `:= createco(`*fn*`,` *size*`)`

This function creates a new coroutine leaving it suspended in the call of `cowait` in the following loop.

```
c := fn(cowait(c)) REPEAT
```

When control is next transfered to the new coroutine, the value passed becomes the result of `cowait` and hence the argument of `fn`. If `fn(..)` returns normally, its result is assigned to `c` which is returned to the parent coroutine by the repeated call of `cowait`. Thus, if `fn` is simple, a call of the coroutine convert the value passed, `val` say, into `fn(val)`. However, in general, `fn` may contain calls of `callco`, `cowait` or `resumeco`, and so the situation is not always quite so simple.

In detail, the implementation of `createco` uses `getvec` to allocate a vector with upper bound `size+6` and initialises its first seven locations ready for the call of `changeco(0,c)` that follows. The state just after this call is shown in figure 3.10. Notice that `cowait(c)` is about to be executed in the environment of the new coroutine, and that this call will cause a return from `createco` in the original coroutine, passing back a pointer to the new coroutine as a result.

`deleteco(`*cptr*`)`

This call takes a coroutine pointer as argument and, after checking that the corresponding coroutine has no parent, deletes it by returning its stack to free store.

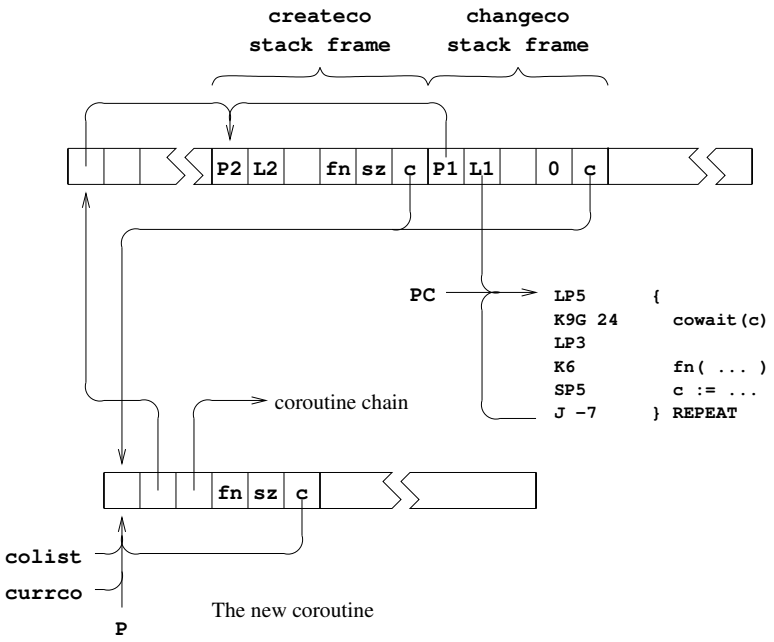


Figure 3.10: The state just after `changeco(0,c)` in `createco`

cptr `:= initco(`*fn*`,` *size*`,` *a*`,...)`

This function is defined in BLIB to provide a convenient method of creating and intialising coroutines. It definition is as follows:

```
LET initco(fn, size, a, b, c, d, e, f, g, h, i, j, k) = VALOF
{ LET cptr = createco(fn, size)
  UNLESS cptr=0 DO callco(cptr, @a)
  RESULTIS cptr
}
```

A coroutine with main function *fn* and given size is created and, if successful, it is initialised by `callco(`*cptr*`, @`*a*`)`. Thus, *fn* should expect a vector containing up to 11 values. Once the coroutine has initialised itself, it should return control to `initco` by means a call of `cowait`. Examples of the use of `initco` can be found in the example that follows.

3.3.12 Hamming's Problem

A following problem permits a neat solution involving coroutines.

> Generate the sequence `1,2,3,4,5,6,8,9,10,12,...` of all numbers divisible by no primes other than 2, 3, or 5".

This problem is attributed to R.W.Hamming. The solution given here shows how data can flow round a network of coroutines. It is illustrated in figure 3.11 in

which each box represents a coroutine and the edges represent `callco/cowait` connections. The end of a connection corresponding to `callco` is marked by `c`, and end corresponding to `cowait` is marked by `w`. The arrows on the connections show the direction in which data moves. Notice that, in `tee1`, `callco` is sometimes used for input and sometimes for output.

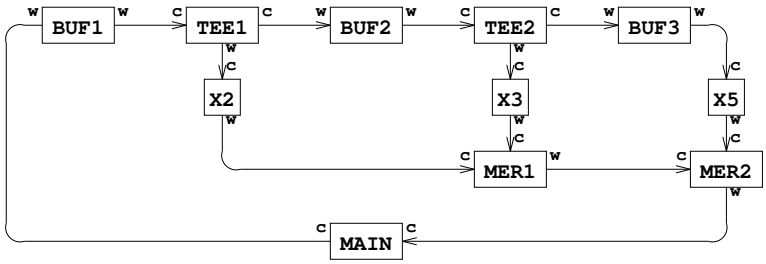


Figure 3.11: Coroutine data flow

The coroutine `buf1` controls a queue of integers. Non-zero values can be inserted into the queue using `callco(buf1,val)`, and values can be extracted using `callco(buf1,0)`. The coroutines `buf2` and `buf3` are similar. The coroutine `tee1` is connected to `buf1` and `buf2` and is designed so that `callco(tee1)` will yield a value extracted from `buf1`, after sending a copy of it to `buf2`. `tee2` similarly takes values from `buf2` passing them to `buf3` and `x3`. Values passing through `x2`, `x3` and `x5` are multiplied by 2, 3 and 5, repectively. `mer1` merges two monotonically increasing streams of numbers produced by `x2` and `x3`. The resulting stream is then merged by `mer2` with the stream produced by `x5`. The stream produced by `mer2` is the required Hamming sequence, each value of which is printed by `main` and then inserted into `buf1`.

The BCPL code for this solution is as follows:

```
GET "libhdr"

LET buf(args) BE    // Body of BUF1, BUF2 and BUF3
{ LET p, q, val = 0, 0, 0
  LET v = VEC 200

  { val := cowait(val)
    TEST val=0 THEN { IF p=q DO writef("Buffer empty*n")
                      val := v!(q REM 201)
                      q := q+1
                    }
               ELSE { IF p=q+201 DO writef("Buffer full*n")
                      v!(p REM 201) := val
                      p := p+1
                    }
  } REPEAT
}
```

```
LET tee(args) BE    // Body of TEE1 and TEE2
{ LET in, out = args!0, args!1
  cowait()          // End of initialisation.

  { LET val = callco(in, 0)
    callco(out, val)
    cowait(val)
  } REPEAT
}

AND mul(args) BE    // Body of X2, X3 and X5
{ LET k, in = args!0, args!1
  cowait()          // End of initialisation.

  cowait(k * callco(in, 0)) REPEAT
}

LET merge(args) BE  // Body of MER1 and MER2
{ LET inx, iny = args!0, args!1
  LET x, y, min = 0, 0, 0
  cowait()          // End of initialisation

  { IF x=min DO x := callco(inx, 0)
    IF y=min DO y := callco(iny, 0)
    min := x<y -> x, y
    cowait(min)
  } REPEAT
}

LET start() = VALOF
{ LET BUF1 = initco(buf,   500)
  LET BUF2 = initco(buf,   500)
  LET BUF3 = initco(buf,   500)
  LET TEE1 = initco(tee,   100, BUF1, BUF2)
  LET TEE2 = initco(tee,   100, BUF2, BUF3)
  LET X2   = initco(mul,   100,    2, TEE1)
  LET X3   = initco(mul,   100,    3, TEE2)
  LET X5   = initco(mul,   100,    5, BUF3)
  LET MER1 = initco(merge, 100,   X2,   X3)
  LET MER2 = initco(merge, 100, MER1,   X5)

  LET val = 1
  FOR i = 1 TO 100 DO { writef(" %i6", val)
                        IF i REM 10 = 0 DO newline()
                        callco(BUF1, val)
                        val := callco(MER2)
                      }

  deleteco(BUF1); deleteco(BUF2); deleteco(BUF3)
  deleteco(TEE1); deleteco(TEE2)
  deleteco(X2); deleteco(X3); deleteco(X5)
  deleteco(MER1); deleteco(MER2)
  RESULTIS 0
}
```

3.3.13 Scaled Arithmetic

The library function `muldiv` makes full precision scaled arithmetic convenient.

res `:= muldiv(`*a*`, `*b*`, `*c*`)`

The result is the value obtained by dividing *c* into the double length product of *a* and *b*, the remainder of this division is left in the global variable `result2`. The result is undefined if it is too large to fit into a single length word or if *c* is zero. In this implementation, the result is also undefined if any of *a*, *b* or *c* is the largest negative integer. As an example, the function defined below calculates the cosine of the angle between two unit vectors in three dimensions using scaled integers to represent numbers with 6 digits after the decimal point.

```
MANIFEST { Unit=1000000 } // Scaling factor for numbers of the
                          // form ddd.dddddd

FUN inprod(v, w) = muldiv(v!0, w!0, Unit) +
                   muldiv(v!1, w!1, Unit) +
                   muldiv(v!2, w!2, Unit)
```

On some processors, such as the Pentium, `muldiv` can be encoded very efficiently in assembly language.

Chapter 4

The Command Language

The Command Language Interpreter (CLI) is a simple interactive interface between the user and the system. It is implemented in BCPL and its source code can be found in `cintcode/sys/CLI.b`. It loads and executes previously compiled programs that are held either in the current directory or a directory specified by the shell variable `BCPLPATH`. The source of the system provided commands can be found in `cintcode/com`. These commands are described in below in Section 4.2. Since any compiled program can be regarded as a command and the command language can be extended easily by the user.

4.1 Bootstrapping

When the Cintcode System is started, control is passed to the interpreter which, after a few initial checks, allocates vectors for the memory of the cintcode abstract machine and the tally vector available for statistics gathering. The cintcode memory is initialised suitably for sub-allocation by `getvec`, which is then used to allocate space for the root node, the initial stack and the initial global vector. The initial state shown in figure 4.1 is completed by loading the object modules `SYSLIB`, `BLIB` and `BOOT`, and initialising the root node, the stack and global vector. Interpretation of cintcode instructions now begins with the Cintcode register `PC`, `P` and `G` set as shown in the figure, and `Count` set to `-1`. The other registers are cleared. The first Cintcode instruction to be executed is the first instruction of the body of the routine `start` defined in `sys/BOOT.b`. Since no return link has been stored into the stack, this call of `start` must not attempt to return in the normal way; however, its execution can still be terminated using `sys(0,0)`.

The global vector and stack shown in figure 4.1 are used by `start` and form the running environment both during initialization and while running the debugger.

The CLI, on the other hand, is provided with a new stack and a separate global vector, thus allowing the debugger to use its own globals freely without interfering with the command language interpreter or running commands. The global vector of 1000 words is allocated for `CLI` and this is shared by the `CLI` program and its running commands. The stack, on the other hand, is used exclusively by the command language interpreter since it creates a coroutine for each command it runs.

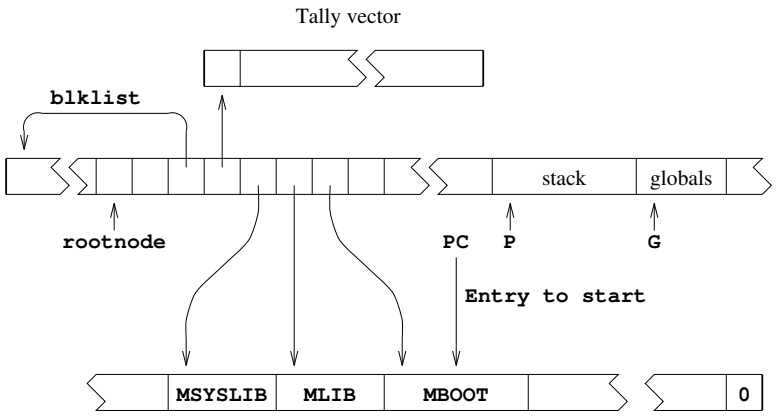


Figure 4.1: The initial state

Control is passed to the `CLI` by means of the call `sys(1,regs)` which recursively enters the intepreter from an initial Cintcode state specified by the vector `regs` in which that `P` and `G` are set to point to the bases of a new stack and a new global vector for `CLI`, respectively, PC is the location of the first instruction of `startcli`, and `count` is set to `-1`. This call of `sys(1,regs)` is embedded in the loop shown below that occurs at the end of the body of `start`.

```
{ LET res = sys(1, regs)  // Call the interpreter
  IF res=0 DO sys(0, 0)
  debug res               // Enter the debugger
} REPEAT
```

At the moment `sys(1,regs)` is first called, only `globsize`, `sys` and `rootnode` have been set in the CLI global vector and so the body of `startcli` must be coded with care to avoid calling global functions before their entry points have be placed in the global vector. Thus, for instance, instead of calling `globin` to initialise the globals defined in `SYSLIB` and `BLIB`, the following code is used:

```
sys(24, rootnode!rtn_syslib)
sys(24, rootnode!rtn_blib)
```

If a fault occurs during the execution of `CLI` or a command that it is running, the call of `sys(1,regs)` will return with the fault code and `regs` will hold the

dumped Cintcode registers. A result of zero, signifying successful completion, causes execution of the Cintcode system to terminate; however, if a non zero result is returned, the debugger in entered by means of the call `debug(res)`. Note that the Cintcode registers are available to the debugger since `regs` is a global variable. When `debug` returns, the `REPEAT`-loop ensures that the command language interpreter is re-entered. The debugger is briefly described in the section 5.

On entry to `startcli`, the coroutine environment is initialised by setting `currco` and `colist` to point to the base of the current stack which is then setup as the root coroutine. The remaining globals are the initialised and the standard input and output streams opened before loading the `CLI` program by means of the following statement:

```
rootnode!rtn_cli := globin(loadseg("CLI"))
```

The command language interpreter is now entered by the call `start()`.

4.2 Commands

This section describes the commands whose source code can be found in `cintcode/com`. Each command is introduced by its name and `rdargs` argument format string.

`abort NUMBER`

The command: `abort` n calls the `BLIB` function `abort` with argument n. If $n = 0$, this will cause a successful return from the Cintcode system. If n is non zero, the interactive debugger is entered with fault code n. The default value for n is `99`. A brief description of the debugger is given in section 5.

```
bcpl    FROM/A,TO/K,VER/K,SIZE/K,TREE/S,NONAMES/S,
            D1/S,D2/S,OENDER/S,EQCASES/S,BIN/S:
```

This invokes the BCPL compiler. The `FROM` argument specified the name of the file to compile. If the `TO` argument is given, the compiler generates code to the specified file. Without the `TO` argument the compiler will just output the OCODE intermediate form to the file `OCODE`. This is used for compiler debugging and cross compilation. The `VER` argument redirects the standard output to a named file. The `SIZE` argument specified the size of the compiler's work space. The default is 40000 words. If the `NONAMES` switch is given the compiler will not include section and function names in the compiled code. These are only useful for debugging. The switches `D1` and `D2` control compiler debugging output. `D1` causes a readable

form of the compiled cintcode to be output. `D2` causes a trace of the internal working of the codegenerator to be output. `D1` and `D2` together causes a slightly more detailed trace of the internal working of the codegenerator to be output. `OENDER` causes code to be generated for a machine with the opposite endianess of the machine on which the compiler is running. `EQCASES` causes all identifiers to be converted to uppercase during compilation. This allows very old BCPL programs to be compiled. `BIN` causes the target cintcode to be in binary rather than ASCII encoded hexadecimal. This is primarily for Windows CE machines where reducing the size of code modules may be important.

`bcplxref FROM/A,TO/K,PAT/K`

This command outputs a cross reference listing of the program given by the `FROM` argument. This consists of a list of all identifiers used in the program each having a list of line numbers where the identifier was used and a letter indicating how the identifier was declared. The letters have the following meanings:

V	Local variable
P	Function or Routine
L	Label
G	Global
M	Manifest
S	Static
F	FOR loop variable

The `TO` argument can be used to redirect the output to a file, and the `PAT` argument supplies a pattern to restrict which names are to be cross referenced. Within a pattern an asterisk will match any sequence of characters, so the pattern `a*b*` will match identifiers such as `ab`, `axxb`or `axbyy`. Upper and lower case letters are equated.

`c` *command-file arguments*

The `c` command allows a file oc commands to be executed as though they had just been typed in. The argument *command-file* gives the name of the file containing the command sequence.

Unless explicitly changed, the characters '`=`', '`<`', '`>`', '`$`' and '`.`' have special meanings within a command command. A dot '`.`' at the start of a line starts a directive which can specify the command command's argument format, or replace one of the special character with an alternative. There are six possible directives as follows:

.KEY or .K	*str*	argument format string
.DEFAULT or .DEF	*key value*	give *key* a default value
.BRA	*ch*	use *ch* instead of <
.KET	*ch*	use *ch* instead of >
.DOLLAR	*ch*	use *ch* instead of $
.DOT	*ch*	use *ch* instead of .

All directives must occur at the start of the command file. The .KEY directive specifies a format string of the form used by rdargs (see page 45) that describes what arguments can follow the command file name. The .DEFAULT directive specifies the default value that a specified key should have if the corresponding argument was omitted. The remaining directives allow the special characters to be changed.

The command sequence occurs after all the directives and may contain items of the form <*key*$*value*> or <*key*> where *key* is one of the keys in the format string and *value* is a default value. Such items are textually replaced by its corresponding argument or a default value. If $*value* is present, this overrides (for this item only) any default that might have been given by a .DEFAULT directive.

casech FROM/A,TO/A,DICT/K,U/S,L/S,A/S

This command systematically processes a BCPL program converting all reserved words to upper case and changing all identifiers to upper case (U), lower case (L, or in the form given by a specified dictionary (DICT).

checksum FROM/A,TO/K

This command calculates a check sum for the file specified by the FROM argument, sending the result to the file specified by the TO argument.

delete ,,,,,,,,,

This command will delete up to ten given files.

detab FROM/A,TO/K,SEP/K

This command copies the file give by the FROM argument to the file given by the TO argument replacing all tab characters by spaces. The tabs are separated by a distance specified by the SEP argument. The default is 8.

echo TEXT,N/S

This command will output its first argument TEXT, if given. The text will be followed by a newline unless the switch N is set.

`edit FROM/A,TO,WITH/K,VER/K,OPT/K`

This command is meant to provide a simple line editor. It used to run on the Tripos Portable Operating System but has not been modified to run on this system.

`fail CODE`

KThis command just returns to the CLI with a completion code given by `CODE`. The default code is 20.

`input TO/A,TERM/K`

This command will copy text from the current input sending it the the file specified by the `AS` argument. The input is terminated by a line starting with `/*` or the value of the `TERM` argument if given.

`interpreter FAST/S,SLOW/S`

This command allows the user to select the fast (`cintasm`) or the slow (`cinterp`) version of the interpreter. If no arguments are given the fast one is selected. It is implemented using `sys(0,-1)` or `sys(0,-2)` as described on page 32.

`join ,,,,,,,,,,,,,,,AS/A/K,CHARS/S`

This command will concatenat several files sending the result to the file specified by the `AS` argument. If the `CHARS` switch is given the files are treated as text files, otherwise they are copied in binary.

`logout`

This command causes an exit from the BCPL Cintcode System, typical returning to an operating system shell.

`map BLOCKS/S,NAMES/S,CODE/S,MAPSTORE/S,TO/K,PIC/S`

This command outputs the state of the Cintcode memory in a form that depends on the arguments given. The output goes to the screen unless a filename is given using the `TO` keyword.

`nlconv FILE,TOUNIX/S,TODOS/S,Q/S`

Thus command replaces the specified file with one in which line endings have been replaced by those appriate for the desination system which is specified by the switches `TOUNIX` (the default) or Windows systems (`TODOS`). The `Q` argument quietens the command.

`prefix PREFIX,UNSET/S`

If the first argument is given, it becomes the current prefix string. If UNSET is specified, the prefix string is unset, and if no argument is given the current prefix is output. This command is implemented using `sys(32,`*prefix*`)` and `sys(33)` described on page 36. See also Section 3.3.5.

`preload ,,,,,,,,,`

This command will preload up to 10 commands into the Cintcode memory. Without arguments it outputs the list of preloaded commands. Preloading improves the efficiency of command execution and is also useful in conjunction with the `stats` command, see below.

`procode FROM,TO/K`

This command converts an OCODE (intermediate code) file specified by `FROM` to a more readable form. If `FROM` is missing it reads from the file `OCODE`. If the `TO` argument is missing it send the result to the screen.

`prompt PROMPT`

This command allows the user to change the prompt string. The prompt is output by the CLI using code of the form:

`writef(`*prompt*`,` *msecs*`)`

where *prompt* is the prompt format string and *msecs* is the time in milliseconds used by the previous command. The default prompt format is: `"%d> "`.

`raster COUNT,SCALE,TO/K,HELP/S`

This command controls the collection of rastering information but only works when the BCPL Cintcode system is running under the rastering interpreter `rasterp`. The implementation uses `sys(27,...)` calls that are described on page 35. If `raster` is given an argument it activates the rastering mechanism. Once rastering is activated information will be written to a raster data file for the duration of the next CLI command. The format of this file is also outlined on page 35.

The `COUNT` argument allows the user to specify how many Cintcode instructions to obey for each raster line. The default is 1000. The `SCALE` argument gives the raster line granularity in bytes per pixel. The default being 12. The `TO` argument specifies the name of the raster data file to be written. The default file name is `RASTER`.

If `raster` is called without any arguments, it closes the raster data file. The raster data file can be processed and converted to Postscript using the `rast2ps`

command described below. Typical use of the `raster` command is following script:

```
raster count 1000 scale 12 to RASTER
bcpl com/bcpl.b to junk
raster
rast2ps fh 18000000 mh 301000
```

This will create the Postscript file `RASTER.ps` for the BCPL compiler compiling itself, similar to that shown in Figure 4.2.

```
rast2ps  FROM,SCALE,TO/K,ML,MH,MG,FL,FH,FG,
              DPI/K,INCL/K,A4/S,A3/S,A2/S,A1/S,A0/S
```

This commands converts a raster data file (written using the `raster` command described above) into a postscript file suitable for printing. There are parameters to control the region to convert, the output paper size and other parameters. It is also possible to posible to include anotations in the resulting picture.

The `FROM` parameter specifies the name of the raster data file. `RASTER` is the default. `SCALE` specifies a magnification as a percentage. The default is 80. The `TO` parameter specifies the name of the postscript file to be generated. `RASTER.ps` is the default. The parameters `ML` and `MH` specify the low and high limits of the address space to be processed. `MG` specifies the separation of the grid line on the memory axis. The defaults are `ML=0 MH=300100` and `MG=100000`. The units are in bytes. The parameters `FL` and `FH` specify the low and high limits of the instruction count axis to be processed. `FG` specifies the separation of the grid line on the memory axis. The defaults are `FL=0 FH=20000000` and `FG=1000000`. `DPI` specified the approximate number of dots per inch used by the output device. The default is 300. `A`*n* specified the output page size. The default is `A4`. The `INCL` parameter specifies the name of a file to be copied into the postscript file. The default is `psincl`. This file allows annotations to be made in the picture. The file `cintcode/psincl` was used to annotate the memory time graph shown in Figure 4.2. This file contains lines such as:

```
F2 setfont
(SYN) 1.1 35 2 PDL
(TRN) 8.1 30 1.7 PUL
(CG) 15.3 36 2.1 PUR
(GET Stream) 0.45 270 1.7 PUL
...
(OCODE Buffer) 13.9 245 2 PDR
% 8.5 150 MVT (HELLO WORLD) SC
F3 setfont
(Self Compilation of the Cintcode BCPL Compiler) TITLE
```

The postscript macros `PDL`, `PUL`, `PUR` and `PDR` draw arrows with specified labels, byte address, instruction count and arrow lengths. The arrow directions

are respectively: down left, Up left, up right and down right. The macro `MVT` moves to the specified position in the graph and `SC` draws a string centered at that position. The `TITLE` macro draws the graph title and `F2` and `F3` are fonts suitable for the labels and title. The resulting postscript file can, of course, be further editied by hand.

`rename FROM/A,TO=AS/A/K`

This will rename the file given by `FROM` to that specified by the `AS` argument.

`stack SIZE`

The command `stack` n causes the size of the coroutine stack allocated for subsequent commands to be n words long. If called without an argument `stack` outputs the current setting.

`stats TO/K,PROFILE/S,ANALYSIS/S`

This command controls the tallying facility which counts the execution of individual Cintcode instructions. If no arguments are given, `stats` turns on tallying by clearing the tally vector and causing tallying to be enabled for the next command to be executed. Subsequent commands are not tallied, making it possible to process the tally vector while it is in a static state. Typical usage of the `stats` command is illustrated below:

`preload queens`	Preload the program to study
`stats`	Enable stats gathering on next command
`queens`	Execute the command to study
`interpreter`	Select the fast interpreter (`cintasm`) `stats` automatically selects the slow one
`stats to STATS`	Send instruction frequencies to file
	or
`stats profile to PROFILE`	Send detailed profile info to file
	or
`stats analysis to ANALYSIS`	Generate statistical analysis to file

`type FROM/A,TO,N/S`

This command will output the file given by the `FROM` argument, sending it to the screen unless the `TO` argument is given. The swirch argument `N` causes line numbers to be added.

`typehex FROM/A,TO/K`

This will convert the file specified by `FROM` in hexadecimal and send the result to the `TO` file if this argument is given.

`unpreload ,,,,,,,,,,ALL/S`

This command will remove preloaded commands from the Cintcode memory. The `ALL` switch will cause all preloaded commands to be removed.

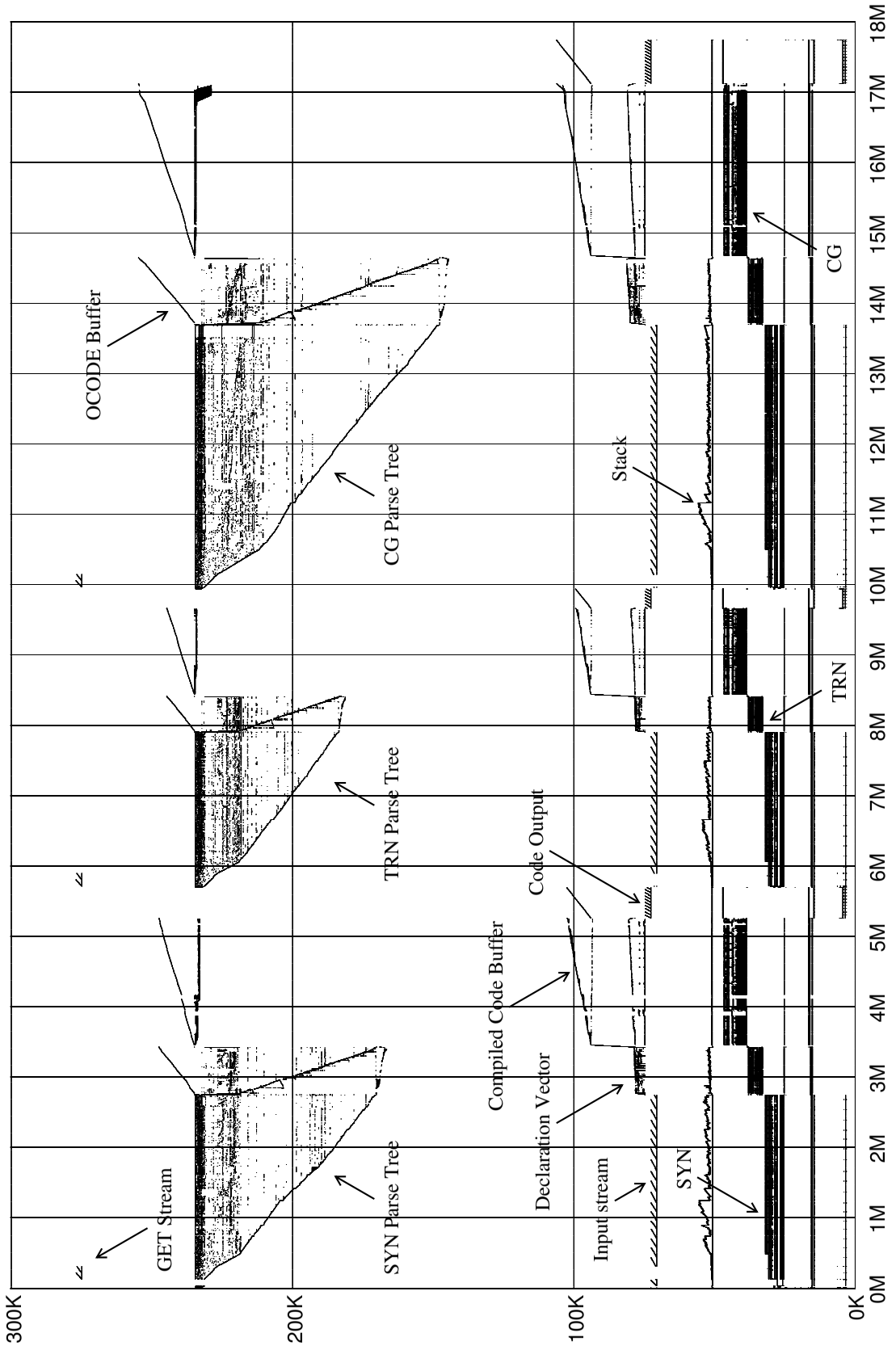


Figure 4.2: Self compilation memory-time graph

Chapter 5

The Debugger

When the Cintcode system starts up, control first passes to `BOOT` which initialises the system and creates a running environment for the command language interpreter (`CLI`). This is run by a recursive invocation of the interpreter and so when faults occur control returns to `BOOT` which then enters an interactive debugger. This allows the user to inspect the state of the registers and memory, and perform other debugging operations on the faulted program. The debugger can also be entered using the `abort` command, as follows:

```
560> abort

!! ABORT 99: User requested
*
```

The asterisk (`*`) is the debugger's prompt character. A brief description of the available debug commands can be display using the query (`?`) command.

```
* ?
?     Print list of debug commands
Gn Pn Rn Vn            Variables
G  P  R  V             Pointers
n #b101 #o377 #x7FF 'c Constants
*e /e %e +e -e |e &e   Dyadic operators
< > !                  Postfixed operators
SGn SPn SRn SVn        Store in variable
=          Print current value
Tn         Print n consecutive locations
$c         Set print style C, D, F, B, O, S, U or X
LL LH      Set Low and High store limits
I          Print current instruction
N          Print next instruction
Q          Quit
B OBn eBn  List, Unset or Set breakpoints
C          Continue execution
X          Equivalent to G4B9C
Z          Equivalent to P1B9C
\          Execute one instruction
,          Move down one stack frame
.          Move to current coroutine
;          Move to parent coroutine
[          Move to first coroutine
]          Move to next coroutine
*
```

The debugger has a current value that can be loaded, modified and displayed. For example:

```
* 12                                  Set the current value to 12
* -2                                  Subtract 2
* *3                                  Multiply by 3
* =          30                       Display the current value
* <                                   Shift left one place
* =          60                       Display the current value
* 12 -2 *3 < =          60           Do it all on one line
*
```

Four areas of memory, namely: the global vector, the current stack frame, the Cintcode register, and 10 scratch variables are easily accessed using the letters `G`, `P`, `R`, `V`, respectively.

```
* 10sv1 11sv2                          Put 10 and 11 in variables 1 and 2
* vt5                                  Display the first 5 variables

V  0:          0          10           11            0            0
*
* v1*50+v2=         511                A calculation using variables
* g0=       1000                      Display global zero (globsize)
* g=        3615                      Display the address of global zero
* ! =       1000                      Indirect and display
* gt10                                 Display the first 10 globals

G  0:       1000     start       stop        sys         clihook
G  5:    GLOB  5     changec       6081        6081           52
*
```

Notice that values that appear to be entry points display the first 7 characters of the function's name. Other display styles can be specified by the commands `$C`, `$D`, `$F`, `$B`, `$O`, `$S`, `$U` or `$X`. These respectively display values as characters, decimal number, in function style (the default), binary, octal, string, unsigned decimal and hexadecimal.

It is possible to display Cintcode instructions using the commands `I` and `N`. For example:

```
* g4=    clihook                      Get the entry to clihook
* n    3340:    K4G  1                Call global 1, incremeting P by 4
* n    3342:    RTN                   Return from the function
*
```

A breakpoint can be set at the first instruction of `clihook` and debugged program re-entered by the following:

```
* g4=    clihook                      Get the entry to clihook
* b9                                  Set break point 9
* c                                   Resume execution
20>
```

The `X` command could have been used since it is a shorhand for `G4B9C`. The function `clihook` is defined in `BLIB` and is called whenever a command is invoked. For example:

```
10> echo ABC                         Invoke the echo command

!! BPT 9:      clihook               Break point hit
   A=          0 B=          0     3340:    K4G  1
*
```

Notice that the values of the Cintcode registers `A` and `B` are displayed, followed by the program counter `PC` and the Cintcode instruction at that point. Single step execution is possible, for example:

```
* \A=          0 B=          0    24228:    LLP  4
* \A=       6097 B=          0    24230:    SP3
* \A=       6097 B=          0    24231:     SP  89
* \A=       6097 B=          0    24233:      L  80
* \A=         80 B=       6097    24235:     SP  90
* \A=         80 B=       6097    24237:    LLL  24272
* \A=       6068 B=         80    24239:     LG  78
* \A=    rdargs  B=       6068    24241:      K  85
* \A=       6068 B=       6068     5480:    LP4
*
```

At this point the first instruction of `rdargs` is about to be executed. Its return address is in `P1`, so a breakpoint can be set to catch the return, as follows:

```
* p1b8
* c

!! BPT 8:        24243
   A=    createc B=          1    24243:   JNE0  24254
*
```

A breakpoint can be set at the start of `sys`, as follows:

```
* g3b1           Set breakpoint 1
* b              Display the currently set of breakpoints
1:      sys
8:        24243
9:      clihook
* 0b8 0b9        Unset breakpoints 8 and 9
* b              Display the remaining breakpoint
1:      sys
*
```

The next three calls of `sys` will be to write the characters `ABC`. The following example steps through these and displays the state of the runtime stack just before the third call, before leaving the debugger.

```
* c

!! BPT 1:      sys
   A=         11 B=         65    21188:    SYS
* c
A
!! BPT 1:      sys
   A=         11 B=         66    21188:    SYS
* c
B
!! BPT 1:      sys
   A=         11 B=         67    21188:    SYS
* .   42844:  Active coroutine     clihook   Size 20000  Hwm   127
      43284:    sys               11          67         312       43228
* ,   43268:    cnslwrf        37772
* ,   43248:    wrch              67          32
* ,   43228:    writes         42915          67
* ,   42888:    start          42904       42912           0     4407873
* ,   42872:    clihook            0
* , Base of stack
* 0b1c                Clear breakpoint 1 and resume
C
210>
```

The following debugging commands allow the coroutine structure to be explored.

Command	Effect
.	Select current coroutine
,	Display next stack frame
;	Select parent coroutine
[	Select first coroutine
]	Select next coroutine

Finally, the command `Q` causes a return from the Cintcode system.

Chapter 6

The design of OCODE

BCPL was designed to be a portable language with a compiler that is easily transferred from machine to machine. To help to achieve this, the compiler is structured as shown in figure 6.1 so that the codegenerator (CG), which is inherently machine dependent, is separated from the rest of the compiler. The front end of the compiler performs syntax analysis producing a parse tree (Tree) which is then translated by the translation phase (TRN) to produce an intermediate form (OCODE) suitable for code generation.

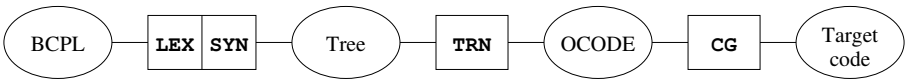


Figure 6.1: The structure of the compiler

6.1 Representation of OCODE

Since OCODE is output by TRN to be read in by CG, there is little need for it to be readable by humans and so is encoded as a sequence of integers which, in the current Cintcode implementation the OCODE is buffered in memory, however the compiler can be made to output a text version to file.

The numerical representation of OCODE can be transformed to the more readable mnemonic form using the `procode` commands, described on page 63. As an example, if the file `test.b` is the following:

```
GET "libhdr"

LET start() BE { LET a, b, c = 1, 0, -1
                 writef("Answer is %n*n", a+b+c)
               }
```

then the command: `bcpl test.b ocode test.ocd` would write the following

file:

```
85 2 94 1 5 115 116 97 114 116 95 3 42 1 42 0 42 -1 92 91 9 43
13 65 110 115 119 101 114 32 105 115 32 37 110 10 40 4 40 3 14
40 5 14 41 74 51 6 97 91 3 103 91 3 90 2 92 76 1 1 1
```

These numbers encode the OCODE statements in a natural way as can be verified by comparing them with the following more readable form of the same statements, generated by the command: `procode test.b`.

```
JUMP L2
ENTRY L1 5  's' 't' 'a' 'r' 't'
SAVE 3 LN 1 LN 0 LN -1 STORE STACK 9
LSTR 13  'A' 'n' 's' 'w' 'e' 'r' ' ' 'i' 's' ' ' '%' 'n' 10
LP 4 LP 3 PLUS LP 5 PLUS LG 74 RTAP 6 RTRN STACK 3
ENDPROC STACK 3 LAB L2 STORE GLOBAL 1 1 L1
```

6.2 The OCODE Abstract Machine

OCODE was specifically designed for BCPL and is a compromise between the desire for simplicity and the conflicting demands of efficiency and machine independence. OCODE is an assembly language for an abstract stack based machine that has a global vector and an area of memory for program and static data as shown in figure 6.2.

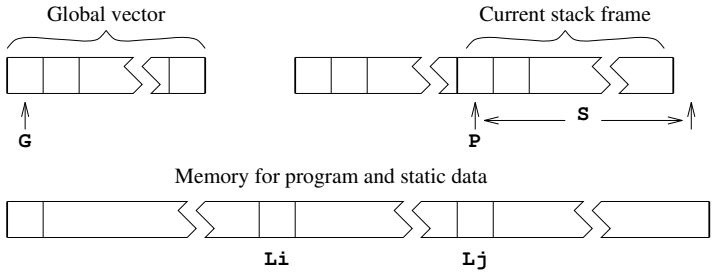


Figure 6.2: The BCPL abstract machine

The global vector is pointed to by the `G` pointer and the current stack frame is pointed to by the `P` pointer. `S` is the size of the current stack frame, and so `P!S` is the first free element of the stack. The value of `S` is always known during compilation and so is not held in a register of the OCODE abstract machine machine. Any assignments to `S` in the description of OCODE statements should be regarded as a specification of `S` for the subsequent statement.

Static variables, tables and string constants are allocated space in the program area and are referenced using labels such as L36 and L92. All global, local and

static variables are of the same size and, on most modern implementations, they hold 32 bit values.

OCODE is normally encoded as a sequence of integers, but for human consumption a more readable form is available. The command `procode` translates the numeric OCODE into this mnemonic form. An OCODE statement consists of a function or directive code possibly followed by operands that are either optionally signed integers, quoted characters or labels of the form Ln where n is a label number. The following are examples of mnemonic OCODE statements:

```
LSTR 5 'H' 'e' 'l' 'l' 'o'
LP 3
GETBYTE
SL L36
```

There are OCODE statements for loading and storing values, for applying expression operators, for procedure handling and flow of control. There are also directives for the allocation of storage and to allow information to be passed to the codegenerator.

6.3 Loading and Storing values

A variables may be local, global or static, and may be accessed in three ways depending on its context, and so there are 9 statements for accessing variables as shown in the following table.

Statement	Meaning
LP n	P!S := P!n; S := S+1
LG n	P!S := G!n; S := S+1
LL Ln	P!S := Ln; S := S+1
LLP n	P!S := @P!n; S := S+1
LLG n	P!S := @G!n; S := S+1
LLL Ln	P!S := @Ln; S := S+1
SP n	P!n := P!S; S := S-1
SG n	G!n := P!S; S := S-1
SL Ln	Ln := P!S; S := S-1

The following tables shows the six statements for loading constants.

Statement	Meaning
LF Ln	P!S := entry point Ln; S := S+1
LN n	P!S := n; S := S+1
TRUE	P!S := TRUE; S := S+1
FALSE	P!S := FALSE; S := S+1
QUERY	P!S := ?; S := S+1
LSTR $nC_1 \ldots C_n$	P!S := "$C_1 \ldots C_n$"; S := S+1

The statements TRUE and FALSE are present to improve portability between machines that use different representations for the integers. For instance, on machines using ones complement or sign and modulus arithmetic, TRUE is not equivalent to LN -1.

Indirect assignments and assignments to elements of word and byte arrays use the statements STIND and PUTBYTE whose meanings are given in table 5.3.

Statement	Meaning
STIND	!(P!(S-1)) := P!(S-2); S := S-2
PUTBYTE	(P!(S-2))%(P!(S-1)) := P!(S-3); S := S-3

Assuming ptr is in global 200, the following assignments:

```
!ptr := 12;  ptr!3 := 99; ptr%3 := 65
```

translate into the following OCODE:

```
LN 12  LG 200  STIND
LN 99  LG 200  LN 3  PLUS    STIND
LN 65  LG 200  LN 3  PUTBYTE
```

6.4 Expression operators

The monadic expression operators only affect the topmost item of the stack and do not change the value of S. They are shown in the next table.

Statement	Meaning
RV	P!(S-1) := ! P!(S-1)
ABS	P!(S-1) := ABS P!(S-1)
NEG	P!(S-1) := - P!(S-1)
NOT	P!(S-1) := $\sim$ P!(S-1)

All dyadic expression operators take two operands from stack replacing them the result and decrementing S by 1. These operators are shown in the following table.

Statement	Meaning
GETBYTE	P!(S-2) := P!(S-2) % P!(S-1)
MULT	P!(S-2) := P!(S-2) * P!(S-1)
DIV	P!(S-2) := P!(S-2) / P!(S-1)
REM	P!(S-2) := P!(S-2) REM P!(S-1)
PLUS	P!(S-2) := P!(S-2) + P!(S-1)
MINUS	P!(S-2) := P!(S-2) - P!(S-1)
EQ	P!(S-2) := P!(S-2) = P!(S-1)
NE	P!(S-2) := P!(S-2) ~= P!(S-1)
LS	P!(S-2) := P!(S-2) < P!(S-1)
GR	P!(S-2) := P!(S-2) > P!(S-1)
LE	P!(S-2) := P!(S-2) <= P!(S-1)
GE	P!(S-2) := P!(S-2) >= P!(S-1)
LSHIFT	P!(S-2) := P!(S-2) << P!(S-1)
RSHIFT	P!(S-2) := P!(S-2) >> P!(S-1)
LOGAND	P!(S-2) := P!(S-2) & P!(S-1)
LOGOR	P!(S-2) := P!(S-2) | P!(S-1)
EQV	P!(S-2) := P!(S-2) EQV P!(S-1)
NEQV	P!(S-2) := P!(S-2) NEQV P!(S-1)

Vector subscription (E_1!E_2 is implemented using `plus` and `RV`.

6.5 Procedures

The design of the OCODE statements for procedure call, save and return have been designed with care to allow code generators as much freedom as possible. The mechanism allows some arguments to be passed in registers if this is required, and the distribution of work between the code for a call and the code at the entry point of a procedure is up to the implementer. In a typical program there are about five calls for each procedure and so there is some incentive to keep the size of the call small by transferring some of the work to the save sequence.

The compilation of a procedure definition generates an OCODE sequence of the following form:

ENTRY Li n C$_1$ ... C$_n$
SAVE s
body of procedure
ENDPROC

Li is the label allocated for the procedure entry point. As a debugging aid, the length of the procedure name is given by n and its characters by the $C_1 \ldots C_n$. The SAVE statement specifies the initial setting of S, which is just the save space size (=3) plus the number of formal parameters. The state of the stack just after procedure entry is shown in figure 6.3.

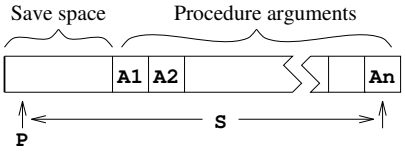


Figure 6.3: The stack frame on procedure entry

The save space is used to hold the previous value of P, the return address and the function entry address. Thus, the first argument of a function is always at position 3 relative to the P pointer. On some older versions of BCPL the size of the save space was different.

The end of the procedure is marked by the ENDPROC statement which is non executable but allows the code generator to keep track of nested procedure definitions. In early versions of OCODE, the first two arguments of ENTRY were interchanged and ENDPROC was given a numerical argument.

The language insists that arguments are laid out in consecutive locations on the stack and that there is no limit to their number. This suggests that a good strategy is to place the values of procedure arguments in the locations they must occupy when the procedure is entered. Thus, a typical call $E(E_1, \ldots, E_n)$ is compiled by first incrementing S to leave room for the save space in the new stack frame, then generate code to evaluate the arguments $E_1, \ldots, E_n$ before generating code for E. The state is then as shown in figure 6.4. Finally, either FNAP k or RTAP k is generated, depending on whether a function or routine call is being compiled. Notice that k is the distance between the old and new stack frames.

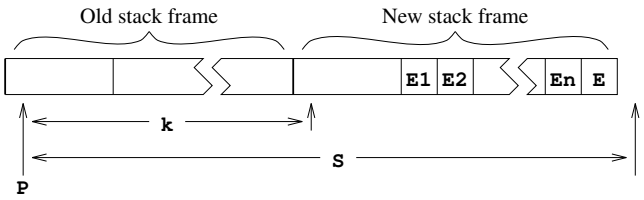


Figure 6.4: The moment of calling E(E1,E2,...En)

The return from a routine is performed by RTRN which restores the previous value of P and resumes execution from the return address. The return from a

function is performed by FNRN just after the function result has been evaluated on the top of the stack. FNRN performs the same action as RTRN, after placing the function result in a special register (A) ready for FNAP to store it in the required location in the previous stack frame.

6.6 Control

The statement LAB Ln set the value of label Ln to the current position in the OCODE program. An unconditional transfer to this label can be performed by the satement JUMP Ln. Conditional jumps inspect the value on the top of the stack P!(S-1). JT Ln will make the jump if it is TRUE, and JF Ln will jump if FALSE. The translation of the command GOTO E is the translation of E followed by the OCODE statement GOTO. It thus takes the destination address from the top of the stack.

If the command RESULTIS E occurs in a context where the value of E is immediately returned as the result of a function, it uses FNRN; but in other contexts, its translation is code to evaluate E followed by a statement of the form RES Ln. This will place the result in the special register (A) and jump to the label Ln, where a statement of the form RSTACK k will be present to accept the value and place it in P!k while setting S to $k + 1$.

The OCODE statement:

$$\texttt{SWITCHON}\ n\ \texttt{L}dK_1L_1 \ldots K_nL_n$$

is used in the compilations of switches. It makes a jump determined by the value on the top of the stack. Its first argument (n) is the number of cases in the switch and the second argument (Ld) is the the default label. K_1 to K_n are the case constants and L_1 to L_n are the corresponding labels.

The FINISH statement is the compilation of the BCPL FINISH command. It is converted into code equivalent to stop(0) by the code generator.

6.7 Directives

Sometimes the size of the stack frame changes other than in the course of expression evaluation. This happens, for instance, when control leaves a block in which local variables were declared. The statement STACK s informs the code generator that the size of the current stack frame is now s.

The STORE statement is used to inform the code generator that the point separating the declarations and body of a block has been reached and that any

anonymous results on the stack are actually initialised local variables and so should be stored in their true stack locations.

Static variables and tables are allocated space in the program area using statements of the form `ITEMN` n, where n is the initial value of the static cell. The elements of table are placed in consecutive locations by consective `ITEMN` statements. A label may be set to the address of a static cell by preceding the `ITEMN` statement by a statement of the form `DATALAB L`n. In earlier versions of OCODE, there was an `ITEML` statement used in the compilation of non global procedures and labels.

The `SECTION` and `NEEDS` directives in a BCPL program translate into `SECTION` and `NEEDS` statements of the form:

$$\texttt{SECTION}\ nC_1 \ldots C_n\ \texttt{NEEDS}\ nC_1 \ldots C_n$$

where C_1 to C_n are the characters of the `SECTION` or `NEEDS` name and n is the length.

The end of an OCODE module is marked by the `GLOBAL` statement which contains information about global procedures and labels. The form of the `GLOBAL` statement is as follows:

$$\texttt{GLOBAL}\ nK_1L_1 \ldots K_nL_n$$

where n is the number of items in the global initialisation list. K_i is the global number and L_i is its label. When a module is loaded its global entry points must be initialised.

6.8 Discussion

A very early version of OCODE used a three address code in which the operands were allowed to be the sum of up to three simple values with a possible indirection. The intention was that reasonable code should be obtainable even when codegenerating one statement at a time. It was soon found more convenient to use an intermediate code that separates the accessing of values from the application of operators. This improved portability by making it possible to implement very simple non optimising codegenerators. Optimising codegenerators could absorb several OCODE statements before emitting compiled code.

The `TRUE` and `FALSE` statements were added in 1968 to improve portability to machines using sign and modulus or one's complement arithmetic. Luckily two's complement arithmetic has now become the norm. Other extension to OCODE,

notably the `ABS`, `QUERY`, `GETBYTE` and `PUTBYTE` statements were added as the corresponding constructs appeared in the language.

In 1980, the BCPL changed slightly to permit position independent code to be compiled. This change specified that non global labels and procedures were no longer variables, and the current version of OCODE reflects this change by the introduction of the `LF` statement and the removal of the old `ITEML` statement that used to allocate static cells for such entry points.

Another minor change in this version of OCODE is the elimination of the `ENDFOR` statement that was provided to fix a problem on 16-bit word addressed machines with more than 64 Kbytes of memory.

Chapter 7

The Design of Cintcode

The original version of Cintcode was a byte stream interpretive code designed to be both compact and capable of efficient interpretation on small 16 bit machines machines based on 8 bit micro processors such as the Z80 and 6502. Versions that ran on the BBC Microcomputer and under CP/M were marketed by RCP Ltd [JR83]. The current version of Cintcode was extended for 32 bit implementations of BCPL and mainly differs from the original by the provision of 32 bit operands and the removal of a size restriction of the global vector.

The Cintcode machine has eight registers as shown in figure 7.1.

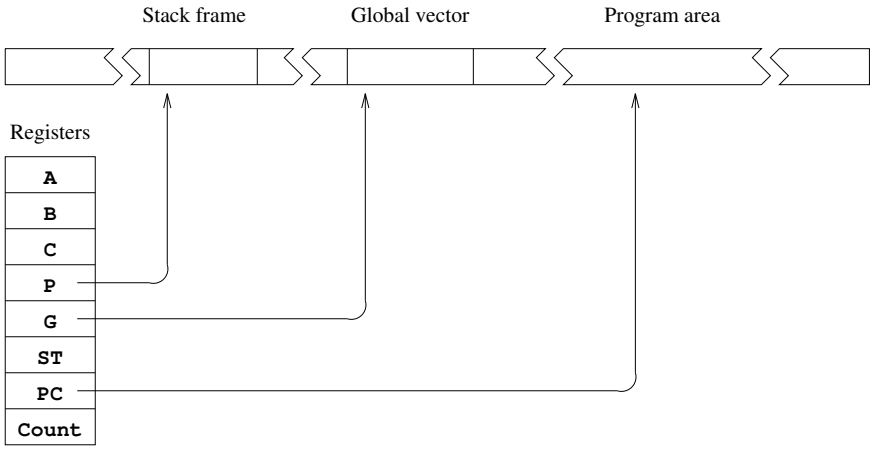


Figure 7.1: The Cintcode machine

The registers `A` and `B` are used for expression evaluation, and `C` is used in in byte subscription. `P` and `G` are pointers to the current stack frame and the global vector, respectively. `ST` was intended as a status register but is currently not used, and `PC` points to the first byte of the next Cintcode instruction to execute. `Count` is a register used by the debugger. While it is positive, `Count` is decremented on

each instruction execution, raising an exception (code 3) on reaching zero. When negative it causes a second (faster) interpreter to be used.

Cintcode encodes the most commonly occurring operations as single byte instructions, using multi-byte instructions for rarer operations. The first byte of an instruction is the function code. Operands of size 1, 2 or 4 bytes imediately follow some function bytes. The two instructions used to implement switches have inline data following the function byte. Cintcode modules also contains static data for stings, integers, tables and global initialisation data.

7.1 Designing for Compactness

To obtain a compact encoding, information theory suggests that each function code should occur with approximately equal frequency. The self compilation of the BCPL compiler, as shown in figure 4.2, was the main benchmark test used to generate frequency information and a summary of how often various operations are used during this test is given in table 7.1. This data was produced using the tallying feature controlled by the `stats` command, described on page 65.

The statistics from different programs vary greatly, so while encoding the common operations really compactly, there is graceful degradation for the rarer cases ensuring that even unusual programs are handled reasonably well. There are, for instance, several one byte instructions for loading small integers, while larger integers are handled using 2, 3 and 5 byte instructions. The intention is that small changes in a source program should cause small small changes in the size of the corresponding compiled code.

Having several variant instructions for the same basic operation does not greatly complicate the compiler. For example the four variants of the `AP` instruction that adds a local variable into register `A` is dealt with by the following code fragment taken from the codegenerator.

```
TEST 3<=n<=12 THEN gen(f_ap0 + n)
              ELSE TEST 0<=n<=255
                   THEN genb(f_ap, n)
                   ELSE TEST 0<=n<=#xFFFF
                        THEN genh(f_aph, n)
                        ELSE genw(f_apw, n)
```

It is clear from table 7.1 that accessing variables and constants requires special care, and that conditional jumps, addition, procedure calls and indirection are also important. Since access to local variables accounts for about a quarter of the operations performed, about this proportion of codes were allocated to instructions concerned with local variables. Local variables are allocated words

Operation	Executions	Static count
Loading a local variable	3777408	1479
Updating a local variable	1965885	1098
Loading a global variable	5041968	1759
Updating a global variable	796761	363
Using a positive constant	4083433	1603
Using a negative constant	160224	93
Conditional jumps (all)	2013013	488
Conditional jumps on zero	494282	267
Unconditional direct jump	254448	140
Unconditional indirect jumps	152646	93
Procedure calls	1324206	1065
Procedure returns	1324204	381
Binary chop switches	43748	12
Label vector switches	96461	17
Addition	2135696	574
Subtraction	254935	111
Other expression operations	596882	74
Loading a vector element	1356315	429
Updating a vector element	591268	137
Loading a byte vector element	476688	53
Updating a byte vector element	405808	29

Table 7.1: Counts from the BCPL self compilation test

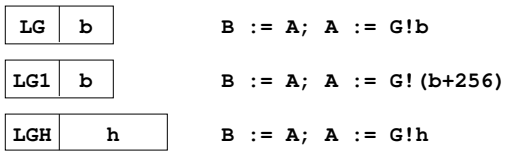
in the stack starting at position 3 relative to the `P` pointer and, as one would expect, small numbered locals are used far more frequently than the others, so operations on low numbered locals often have single byte codes.

Although not shown here, other statistics, such as the distribution of relative addressing offsets and operand values, influenced the design of Cintcode.

7.1.1 Global Variables

Global variables are referenced as frequently as locals and therefore have many function codes to handle them. The size of the global vector in most programs is less than 512, but Cintcode allows this to be as large are 65536 words. Each op-

eration that refers to a global variable is provided with three related instructions. For instance, the instructions to load a global into register `A` are as follows:



Here, `b` and `h` are unsigned 8 and 16 bit values, respectively.

7.1.2 Composite Instructions

Compactness can be improved by combining commonly occurring pairs (and triples) of operations into a single instructions. Many such composite instructions occur in Cintcode; for instance, `AP3` adds local `3` to the `A` register, and `L1P6` will load `v!1` into register `A`, assuming `v` is held in local `6`.

7.1.3 Relative Addressing

A relative addressing mechanism is used in conditional and uncoditional jumps and the instructions: `LL`, `LLL`, `SL` and `LF`. Al these instructions refer to locations within the code and are optimised for small relative distances. To simplify the codegenerator all relative addressing instructions are 2 bytes in length. The first being the function code and the second being an 8 bit relative address.

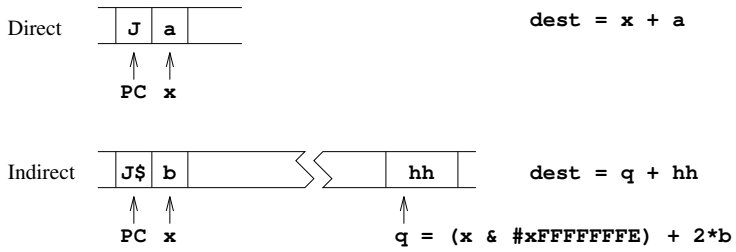


Figure 7.2: The relative addressing mechanism

All relative addressing instructions have two forms: direct and indirect, depending on the least significant bit of the function byte. The details of both relative address calculations are shown in figure 7.2, using the instructions `J` and `J$` as examples. For the direct jump (`J`), the operand (`a`) is a signed byte in the range -128 to +127 which is added to the address (`x`) of the operand byte to give the destination address (`dest`). For the indirect jump, `J$`, the operand (`b`) is an unsigned byte in the range 0 to 255 which is doubled and added to the rounded

version of `x` to give the address (`q`) of a 16 bit signed value `hh` which is added to `q` to give the destination address (`dest`).

The compiler places the resolving half word as late as possible to increase the chance that it can be shared by other relative addressing instructions to the same desination, as could happen when several ENDCASE statements occur in a large SWITCHON command. The use of a 16 bit resolving word places a slight restriction on the maximum size of relative references. Any Cintcode module of less than 64K bytes will have no problem.

7.2 The Cintcode Instruction Set

The resulting selection of function codes is shown in Table 7.2 and they are described in the sections that follow. In the remaining sections of this chapter the following conventions hold:

Symbol	Meaning
n	An integer encoded in the function byte.
Ln	The one byte operand of a relative addressing instruction.
b	An unsigned byte, range $0 \leq b \leq 255$.
h	An unsigned halfword, range $0 \leq h \leq 65535$.
w	A signed 32 bit word.
filler	Optional filler byte to round up to a 16 bit boundary.
`A`	The Cintcode `A` register.
`B`	The Cintcode `B` register.
`C`	The Cintcode `C` register.
`P`	The Cintcode `P` register.
`G`	The Cintcode `G` register.
`PC`	The Cintcode `PC` register.

7.2.1 Byte Ordering and Alignment

A Cintcode module is a vector of 32 bit words containing the compiled code and static data of a section of program. The first word of a module holds its size in words that is used as a relative address to the end of the module where the global initialisation data is placed. The last word of a module holds the highest referenced global number, and working back, there are pairs of words giving the global number and relative entry address of each global function or label defined in the module. A relative address of zero marks the end of the initialisation data. See section 6.3 for more details.

	0	32	64	96	128	160	192	224
0	-	K	LLP	L	LP	SP	AP	A
1	-	KH	LLPH	LH	LPH	SPH	APH	AH
2	BRK	KW	LLPW	LW	LPW	SPW	APW	AW
3	K3	K3G	K3G1	K3GH	LP3	SP3	AP3	L0P3
4	K4	K4G	K4G1	K4GH	LP4	SP4	AP4	L0P4
5	K5	K5G	K5G1	K5GH	LP5	SP5	AP5	L0P5
6	K6	K6G	K6G1	K6GH	LP6	SP6	AP6	L0P6
7	K7	K7G	K7G1	K7GH	LP7	SP7	AP7	L0P7
8	K8	K8G	K8G1	K8GH	LP8	SP8	AP8	L0P8
9	K9	K9G	K9G1	K9GH	LP9	SP9	AP9	L0P9
10	K10	K10G	K10G1	K10GH	LP10	SP10	AP10	L0P10
11	K11	K11G	K11G1	K11GH	LP11	SP11	AP11	L0P11
12	LF	S0G	S0G1	S0GH	LP12	SP12	AP12	L0P12
13	LF$	L0G	L0G1	L0GH	LP13	SP13	XPBYT	S
14	LM	L1G	L1G1	L1GH	LP14	SP14	LMH	SH
15	LM1	L2G	L2G1	L2GH	LP15	SP15	BTC	MDIV
16	L0	LG	LG1	LGH	LP16	SP16	NOP	CHGC0
17	L1	SG	SG1	SGH	SYS	S1	A1	NEG
18	L2	LLG	LLG1	LLGH	SWB	S2	A2	NOT
19	L3	AG	AG1	AGH	SWL	S3	A3	L1P3
20	L4	MUL	ADD	RV	ST	S4	A4	L1P4
21	L5	DIV	SUB	RV1	ST1	XCH	A5	L1P5
22	L6	REM	LSH	RV2	ST2	GBYT	RVP3	L1P6
23	L7	XOR	RSH	RV3	ST3	PBYT	RVP4	L2P3
24	L8	SL	AND	RV4	STP3	ATC	RVP5	L2P4
25	L9	SL$	OR	RV5	STP4	ATB	RVP6	L2P5
26	L10	LL	LLL	RV6	STP5	J	RVP7	L3P3
27	FHOP	LL$	LLL$	RTN	GOTO	J$	ST0P3	L3P4
28	JEQ	JNE	JLS	JGR	JLE	JGE	ST0P4	L4P3
29	JEQ$	JNE$	JLS$	JGR$	JLE$	JGE$	ST1P3	L4P4
30	JEQ0	JNE0	JLS0	JGR0	JLE0	JGE0	ST1P4	-
31	JEQ0$	JNE0$	JLS0$	JGR0$	JLE0$	JGE0$	-	-

Table 7.2: The Cintcode function codes

The compiler can generate code for either a big- or little-endian machine. These differ only in the byte ordering of bytes within words. For a little endian machine, the first byte of a 32 bit word is at the least significant end, and on a big-endian machine, it is the most significant byte. This affect the ordering of bytes in 2 and 4 byte immediate operands, 2 byte relative address resolving words, 4 byte static quantities and global initialisation data. Resolving words are aligned on 16 bit boundaries relative to the start of the module, and 4 byte statics values are aligned on 32 bit boundaries. The 2 and 4 byte immediate operands are not aligned.

For efficiency reasons, the byte ordering is chosen to suit the machine on which the code is to be interpreted. The compiler option `OENDER` causes the BCPL compiler to compile code with the opposite endianess to that of the machine on which the compiler is running, see the description of the `bcpl` command on page 59.

7.2.2 Loading values

The following instructions are used to load constants, variables, the addresses of variables and function entry points. Notice that all loading instructions save the old value of register `A` in `B` before updating `A`. This simplifies the translation of dyadic expression operators.

`L`n	$0 \le n \le 10$	`B := A; A :=` n
`LM1`		`B := A; A := -1`
`L` b		`B := A; A :=` b
`LH` h		`B := A; A :=` h
`LMH` h		`B := A; A :=` $-h$
`LW` w		`B := A; A :=` w

These instructions load integer constants. Constants are in the range `-1` to `10` are the most common and have single byte instructions. The other cases use successively larger instructions.

`LP`n	$3 \le n \le 16$	`B := A; A := P!`n
`LP` b		`B := A; A := P!`b
`LPH` h		`B := A; A := P!`h
`LPW` w		`B := A; A := P!`w

These instructions load local variables and anonymous results addressed relative to `P`. Offsets in the range 3 to 16 are the most common and use single byte instructions. The other cases use succesively larger instructions.

`LG` b	`B := A; A := G!`b
`LG1` b	`B := A; A := G!(`$b+256$`)`
`LGH` h	`B := A; A := G!`h

`LG` loads the value of a global variables in the range 0 to 255, `LG1` load globals in the range 256 to 511, and `LGH` can load globals up to 65535. Global numbers must be in the range 0 to 65535.

`LL` Ln	`B := A; A :=` variable Ln
`LL$` Ln	`B := A; A :=` variable Ln
`LF` Ln	`B := A; A :=` entry point Ln
`LF$` Ln	`B := A; A :=` entry point Ln

`LL` loads the value of a static variable and `LF` loads the entry address of a function, routine or label in the current module.

`LLP` b	`B := A; A := @P!`b
`LLPH` h	`B := A; A := @P!`h
`LLPW` w	`B := A; A := @P!`w
`LLG` b	`B := A; A := @G!`b
`LLG1` b	`B := A; A := @G!(`$b+256$`)`
`LLGH` h	`B := A; A := @G!`h
`LLL` Ln	`B := A; A := @(variable` Ln`)`
`LLL$` Ln	`B := A; A := @(variable` Ln`)`

These instructions load the BCPL ponters to local, global and static variables.

7.2.3 Indirect Load

`GBYT`		`A := B%A`
`RV`		`A := A!0`
`RV`n	$1 \le n \le 6$	`A := A!`n
`RVP`n	$3 \le n \le 7$	`A := P!`n`!A`
`L0P`n	$3 \le n \le 12$	`B := A; A := P!`n`!0`
`L1P`n	$3 \le n \le 6$	`B := A; A := P!`n`!1`
`L2P`n	$3 \le n \le 5$	`B := A; A := P!`n`!2`
`L3P`n	$3 \le n \le 4$	`B := A; A := P!`n`!3`
`L4P`n	$3 \le n \le 4$	`B := A; A := P!`n`!4`
`L`n`G` b	$0 \le n \le 2$	`B := A; A := G!`b`!`n
`L`n`G1` b	$0 \le n \le 2$	`B := A; A := G!(`b`+256)!`n
`L`n`GH` h	$0 \le n \le 2$	`B := A; A := G!`h`!`n

These instructions are used in the implementation of byte and word indirection operators `%` and `!` in right hand contexts.

7.2.4 Expression Operators

NEQ	A := -A
ABS	A := ABS A
NOT	A := ~A

These instructions implement the three monadic expression operators.

MUL	A := B * A
DIV	A := B / A
REM	A := B REM A
ADD	A := B + A
SUB	A := B - A
LSH	A := B << A
RSH	A := B >> A
AND	A := B & A
OR	A := B | A
XOR	A := B NEQV A

These instructions provide for all the normal arithmetic and bit pattern dyadic operators. The instructions DIV and REM generate exception 5 if the divisor is zero. Evaluation of relational operators in non conditional contexts involve conditional jumps and the FHOP instruction, see page 96. Addition is the most frequently used arithmetic operation and so there are various special instructions improve its efficiency.

An	$1 \leq n \leq 5$	A := A + n
Sn	$1 \leq n \leq 4$	A := A - n
A b		A := A + b
AH h		A := A + h
AW w		A := A + w
S b		A := A - b
SH h		A := A - h

These instructions implement addition and subtraction by a constant integer amounts. There are single byte instructions for incrementing by 1 to 5 and decremented by 1 to 4. For other values longer instructions are available.

APn	$3 \leq n \leq 12$	A := A + P!n
AP b		A := A + P!b
APH h		A := A + P!h
APW w		A := A + P!w
AG b		A := A + G!b
AG1 b		A := A + G!(b+1)
AGH h		A := A + G!b

These instructions allow local and global variables to be added to A. Special instructions for addition by static variables are not provided, and subtraction by a variable is not common enough to warrant special treatment.

7.2.5 Simple Assignment

SPn	$3 \leq n \leq 16$	P!n := A
SP b		P!b := A
SPH h		P!h := A
SPW w		P!w := A
SG b		G!b := A
SG1 b		G!(b+256) := A
SGH h		G!h := A
SL Ln		variable Ln := A
SL$ Ln		variable Ln := A

These instructions are used in the compilation of assignments to named local, global and static variables. The SP instructions are also used to save anonymous results and to layout function arguments.

7.2.6 Indirect Assignment

PBYT		B%A := C
XPBYT		A%B := C
ST		A!0 := B
STn	$1 \leq n \leq 3$	A!n := B
ST0Pn	$3 \leq n \leq 4$	P!n!0 := A
ST1Pn	$3 \leq n \leq 4$	P!n!1 := A
STPn	$3 \leq n \leq 5$	P!n!A := B
S0G b		G!b!0 := A
S0G1 b		G!(b+256)!0 := A
S0GH h		G!h!0 := A

These instructions are used in assignments in which % or ! appear as the leading operator on the left hand side.

7.2.7 Procedure calls

At the moment a function or routine is called the state of the stack is as shown in figure 7.3. At the entry point of a function or routine the first argument, if any, will be in register A and in memory P!3.

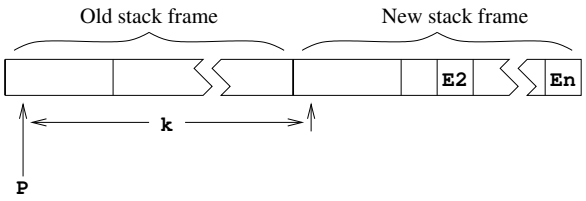


Figure 7.3: The moment of calling `E(E1,E2,...En)`

Kn	$3 \leq n \leq 11$
K b	
KH h	
KW w	

These instructions call the function or routine whose entry point is in `A` and whose first argument (if any) is in `B`. The new stack frame at position `k` relative to `P` where `k` is n, b, h or w depending on which instruction is used. The effect of these instructions is as follows:

```
P!k := P    // Save the old P pointer
P   := P+k  // Set its new value
P!1 := PC   // Save the return address
PC  := A    // Set PC to the entry point
P!2 := PC   // Save it in the stack for debugging
A   := B    // Put the first argument in A
P!3 := A    // Save it in the stack
```

As can be seen, three words of link information (the old `P` pointer, the return address and entry address) are stored in the base of the new stack frame.

KnG b	$3 \leq n \leq 11$
KnG1 b	$3 \leq n \leq 11$
KnGH h	$3 \leq n \leq 11$

These instructions deal with the common situation where the entry point of the function is in the global vector and the stack increment is in the range 3 to 11. The global number `gn` is b, $b+256$ or h depending on which function code is used and stack increment `k` is n. The first argument (if any) is in `A`. The effect of these instructions is as follows:

```
P!k := P    // Save the old P pointer
P   := P+k  // Set its new value
P!1 := PC   // Save the return address
PC  := G!gn // Set the new PC value from the global value
P!2 := PC   // Save it in the stack for debugging
P!3 := A    // Save the first argument in the stack
```

```
RTN
```

This instruction causes a return from the current function or routine using the previous P pointer and the return address held in P!0 and P!1. The effect of the instruction is as follows:

```
PC  := P!1 // Set PC to the return address
P   := P!0 // Restore the old P pointer
```

When returning from a function the result will be in A.

7.2.8 Flow of Control and Relations

The following instructions are used in the compilation of conditional and unconditional jumps, and relational expressions. The symbol *rel* denotes EQ, NE, LS, GR, LE or GE indicating the relation being tested.

J *Ln*	PC := *Ln*
J$ *Ln*	PC := *Ln*
J*rel* *Ln*	IF B *rel* A DO PC := *Ln*
J*rel*$ *Ln*	IF B *rel* A DO PC := *Ln*
J*rel*0 *Ln*	IF A *rel* 0 DO PC := *Ln*
J*rel*0$ *Ln*	IF A *rel* 0 DO PC := *Ln*

The destinations of these jump instructions are computed using the relative addressing mechanism described in section 7.1.3. Notice than when the comparison is with zero, A holds the left operand of the relation.

```
GOTO            PC := A
```

This instruction is only used in the compilation of the GOTO command.

```
FHOP            A := 0; PC := PC+1
```

The FHOP instruction is only used in the compilation of relational expressions in non conditional contexts as in the compilation. The assignment: `x := y < z` is typically compiled as follows:

LP4	Load y
LP5	Load z
JLS 2	Jump to the LM1 instruction if y<z
FHOP	A := FALSE; and hop over the LM1 instruction
LM1	A := TRUE
SP3	Store in x

7.2.9 Switch Instructions

The instructions are used to implement switches are `SWL` and `SWB`, switching on the value held in `A`. They both assume that all case constants are in the range 0 to 65535, with the compiler taking appropriate action when this constraint is not satisfied.

`SWL` *filler* n *dlab* $L_0 \ldots L_{n-1}$

This instruction is used when there are sufficient case constants all within a small enough range. It performs the jump by selecting an element from a vector of 16 bit resolving half words. The quantities n, *dlab*, and L_0 to L_{n-1} are 16 bit half words, aligned on 16 bit boundaries by the option filler byte. If `A` is in the range 0 to $n-1$ it uses the appropriate resolving half word $L_{\texttt{A}}$, otherwise it uses the resolving half word *dlab* to jump to the default label. See Section 7.1.3 for details on how resolving half words are interpreted.

`SWB` *filler* n *dlab* K_1 L_1 $\ldots K_n$ L_n

This instruction is used when the range of case constants is too large for `SWL` to be economical. It performs the jump using a binary chop strategy. The quantities n, *dlab*, K_1 to K_n and L_1 to L_n are 16 bit half words aligned on 16 bit boundaries by the option filler byte. This instruction successively tests `A` with the case constants in the balanced binary tree given in the instruction. The tree is structured in a way similar to that used in heapsort with the children of the node at position i at positions $2i$ and $2i+1$. References to nodes beyond n are treated as null pointers. Within this tree, K_i is greater than all case constants in the tree rooted at position $2i$, and less than those in the tree at $2i+1$. The search starts at position 1 and continues until a matching case constant is found or a null pointer is reached. If `A` is equal to some K_i then `PC` is set using the resolving half word L_i, otherwise it uses the resolving half word *dlab* to jump to the default label. See Section 7.1.3 for details on how resolving half words are interpreted.

The use of this structure is particularly good for the hand written machine code interpreter for the Pentium where there are rather few central registers. Cunning use can be made of the add with carry instruction (`adcl`). In the following fragment of code, `%esi` points to n, `%eax` holds i and `A` is held in `%eab`.

There is a test elsewhere to ensure that A is in the range 0 to 65535.

```
swb1:  cmpw (%esi,%eax,4),%bx ; { compare A with Ki
       je swb3                ;   Jump if A=Ki
       adcl                   ;   IF A>Ki THEN i := 2i
                              ;           ELSE i := 2i+1
       cmpw (%esi),%ax        ;
       jle swb1               ; } REPEATWHILE i<=n
```

The compiler ensures that the tree always has at least 7 nodes allowing the code can be further improved by preceeding this loop with two copies of:

```
       cmpw (%esi,%eax,4),%bx ;   compare Ki with A
       je swb3                ;   Jump if match found
       adcl                   ;   IF A>Ki THEN i := 2i
                              ;           ELSE i := 2i+1
```

The above code is a great improvement on any straightforward implementation of the standard binary chop mechanism.

7.2.10 Miscellaneous

```
XCH            Exchange A and B
ATB            B := A
ATC            C := A
BTC            C := B
```

These instructions are used move values between register A, B and C.

NOP

This instruction has no effect.

SYS

This instruction is used in body of the hand written library routine sys. If A is zero then the interpreter returns with exception code P!4.

If A is -1 it set register count to P!4, setting A to the previous value of count. Changing the value of count may change which of the two interpreters is used. For more details see Section 3.2.2.

Otherwise, it performs a system operation returning the result in A. In the C implementation of the interpreter this is done by the following code:

```
c = dosys(p, g);
```

MDIV

This instruction is used as the one and only instruction in the body of the hand written library routine muldiv, see Section 3.3.13. It divides P!5 into the double length product of P!3 and P!4 placing the result in A and the remainder in the global variable result2. It then performs a function return (RTN). Its effect is as follows:

```
A            := <the result>
G!Gn_result2 := <the remainder>
PC           := P!1              // PC      := P!1
P            := P!0              // P       := P!0
```

CHGCO

This instruction is used in the implementation of coroutines. It is the one and only instruction in the body of the hand written library routine chgco. Its effect, which is rather subtle, is as follows:

```
G!Gn_currco!0 := P!0     // !currco := !P
PC            := P!1     // PC      := P!1
G!Gn_currco   := P!4     // currco  := cptr
P             := P!4!0   // P       := !cptr
```

BRK

This instruction is used by the debugger in the implementation of break points. It causes the interpreter to return with exception code 2.

7.2.11 Undefined Instructions

These instructions have function codes 0, 1, 232, 254 and 255, and they each cause the interpreter to return with exception code 1.

7.2.12 Corruption of B

To improve the efficiency of some hand written machine code interpreters, the following instructions are permitted to corrupt the value held in B:

K KH KW K*n* K*n*G K*n*G1 K*n*GH
SWL SWB MDIV CHGCO

All other instructions either set B explicitly or leave its value unchanged.

7.2.13 Exceptions

When an exception occurs, the interpreter saves the Cintcode registers in its register vector and yields the exception number as result. For exceptions caused by non existent instructions, BRK, DIV or REM the program counter is left pointing to the offending instruction. For more details see the description of `sys(1,` *regs*) on page 32.

Chapter 8

Installation

The implementation of BCPL described in this report is is available free via my Home Page [Ric] to individuals for private use and to academic institutions. If you install the system, please send me a message (to `mr@cl.cam.ac.uk`) so I can keep a record of who is interested in it.

This implementation is designed to be machine independent being based on an interpreter written in C. There are, however, hand written assembly language versions of the interpreter for several architectures (including i386, MIPS, ALPHA and Hitachi SH3). For Windows 95/98/NT and Windows CE there are precompiled `.exe` files, but for all the other architectures it is necessary to rebuild the system.

The simplest installation is for Linux machines.

8.1 Linux Installation

This section describes how to install the BCPL Cintcode System on an IBM PC running Linux.

- First create a directory named `BCPL` and copy either `bcpl.targz` or `bcpl.zip` into it. They are available (free) via my home page [Ric] and both contain the same set of packed files and directories.
- Either unpack `bcpl.targz` by:

```
tar zxvf bcpl.targz
```

  or unpack `bcpl.zip` using:

```
unzip -v bcpl.zip
```

This will create the directories `cintcode`, `bcplprogs` and `natbcpl`. The directory `cintcode` contains all the source files of the BCPL Cintcode System, `bcplprogs` contains a collection of demonstration programs, and `natbcpl` contains a version of BCPL that compiles into native code (for Intel and ALPHA machines.

- Now change directory to `cintcode`.

```
cd cintcode
```

- Re-build enter the BCPL system:

```
make
```

  This should generate output that ends with:

```
BCPL Cintcode System
0>
```

  indicating that the Command Langage Interpreter has been successfully entered.

- It is now necessary to recompile all the system software and commands. This is done by typing:

```
c compsys
```

- Now try out a few commands, eg:

```
echo hello
bcpl com/echo.b to junk
junk hello
map pic
logout
```

- The BCPL programs that are part of the system are: `BOOT.b`, `BLIB.b` and `CLI.b`. These reside in `BCPL/cintcode/sys` and can be compiled by the following commands (in the BCPL Cintcode System).

```
c bs BOOT
c bs BLIB
c bs CLI
```

  The standard commands are in `BCPL/cintcode/com` may be compiled using `bc`.

```
c bc echo
c bc abort
c bc logout
c bc stack
c bc map
c bc prompt
```

  Read the documentation in `cintcode/doc` and any `README` files you can find. A log of recent changes can be found in `cintcode/doc/changes`. A postscript version of the current version of this BCPL manual is available from my home page. There is a demonstration script of commands in `cintcode/doc/notes`.

- In order to use the BCPL Cintcode System from another directory it is necessary to define the shell variable `BCPLPATH` to be the absolute file name of the cintcode directory and add this directory to your `PATH`, before entering the BCPL Cintcode system. On Linux, this can be done by:

```
export BCPLPATH=/home/mr/distribution/BCPL/cintcode
export PATH=$PATH:$BCPLPATH
```

  The shell variable `BCPLPATH` is used when loading Cintcode object modules, reading BCPL header files and command-command files read by the `c` command.

- To compile and run a demo program such as `bcplprogs/demos/queens.b`:

```
cd ../bcplprogs/demos
cinterp
c b queens
queens
```

8.2 Command Line Arguments

The command `cinterp` that invokes the Cintcode interpreter can be given arguments to control memory allocation. These are:

`-m` n Set the cintcode memory size to $1000n$ words
`-t` n Set the tally vector size to $1000n$ words
`-h` Output some help information

The rastering version of the interpreter `rasterp` can receive the same arguments.

8.3 Installation on Other Machines

Carry out steps 1 to 4 above. In the directory `BCPL/cintcode/sys` you will find directories for different architectures, e.g. ALPHA, MIPS, SUN4, SPARC, MSDOS, MAC, OS2, BC4, Win32, CYGWIN32 and shWinCE. These contain files that are architecture (or compiler) dependent, typically including `cintasm.s` (or `cintasm.asm`). For some old versions of Linux, it is necessary to change `_dosys` to `dosys` (or vice-versa) in the file `sys/LINUX/cintasm.s`.

Edit Makefile (typically by adding and removing comment symbols) as necessary for your system/machine and then execute make in the cintcode directory, e.g:

```
make
```

Variants of the above should work for the other architectures running Unix.

8.4 Installation for Windows 95/98/NT

The files `cinterp.exe` and `rasterp.exe` are included in the standard distribution and should work under these operating systems just by typing the command:

```
cinterp
```

I recommend using the GNU development tools and utilities for Windows 95/98/NT/etc that are available from `http://sourceware.cygnus.com/cygwin/`.

Edit the `cintcode/Makefile` to comment out the LINUX version

```
#CC = gcc -O9 -DforLINUX
#SYSM = ../cintcode/sys/LINUX
#CINTASM = cintasm.o
#ENDER = LITENDER
```

and enable the CYGWIN32 version

```
CC = gcc -O9 -DforCYGWIN32
CINTASM = cintasm.o
SYSM = ../cintcode/sys/CYGWIN32
ENDER = LITENDER
```

Then type:

```
make
```

This should recompile the system and create the executable `cinterp.exe`.

Remember to include the cintcode directory in your `PATH` and `BCPLPATH` shell variables, so that the cintcode system can be run in any directory.

Careful inspection of the Makefile and directories in `cintcode/sys` will show that versions also exist that use Microsoft C++ 5.0 and Borland C4.0, but these are likely to be out of date and their use is not recommended.

8.5 Installation for Windows CE2.0

A version of the BCPL Cintcode System is available for handheld machines running Windows CE version 2.0. For instalaation details see the file `cintcode/sys/shWinCE/README`. This system provides ascrollable window for interaction with the CLI. It also provides a simple graphical facilities using a graphics window. The system has only been tested on an HP 620LX handheld machine.

8.6 The Native Code Version

A BCPL native mode system for 386/486/Pentium based machines is in directory `MCPL/native`. It can be re-built and test by changing to the directory `BCPL/natbcpl` and running `make`.

A version (64 bit) for the DEC Alpha is also available. To re-build this it is necessary to comment out the lines for LINUX and uncomment the lines for the ALPHA in `Makefile`, before running `make`.

Chapter 9

Example Programs

9.1 Coins

The following program prints out how many different ways a sum of money can be composed from coins of various denominations.

```
GET "libhdr"

LET coins(sum) = c(sum, (TABLE 200, 100, 50, 20, 10, 5, 2, 1, 0))

AND c(sum, t) = sum<0 -> 0,
                sum=0 -> 1,
                !t=0  -> 0,
                c(sum, t+1) + c(sum-!t, t)

LET start() = VALOF
{ writes("Coins problem*n")
  t(0); t(1); t(2); t(5); t(21); t(100); t(200)
  RESULTIS 0
}

AND t(n) BE writef("Sum = %i3  number of ways = %i6*n", n, coins(n))
```

9.2 Primes

The following program prints out a table of all primes less than 1000, using the sieve method.

```
GET "libhdr"

GLOBAL { count: ug  }

MANIFEST { upb = 999  }

LET start() = VALOF
{ LET isprime = getvec(upb)
   count := 0
   FOR i = 2 TO upb DO isprime!i := TRUE  // Until proved otherwise.

   FOR p = 2 TO upb IF isprime!p DO
   { LET i = p*p
      UNTIL i>upb DO { isprime!i := FALSE; i := i + p }
      out(p)
   }

   writes("*nend of output*n")
   freevec(isprime)
   RESULTIS 0
}

AND out(n) BE
{ IF count REM 10 = 0 DO newline()
   writef(" %i3", n)
   count := count + 1
}
```

9.3 Queens

The following program calculates the number of ways n queens can be placed on a $n \times n$ chess board without any two occupying the same row, column or diagonal.

```
GET "libhdr"

GLOBAL { count:200; all:201  }

LET try(ld, row, rd) BE TEST row=all

                        THEN count := count + 1

                        ELSE { LET poss = all & ~(ld | row | rd)
                               UNTIL poss=0 DO
                               { LET p = poss & -poss
                                 poss := poss - p
                                 try(ld+p << 1, row+p, rd+p >> 1)
                               }
                             }
```

```
LET start() = VALOF
{ all := 1

  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("Number of solutions to %i2-queens is %i5*n", i, count)
    all := 2*all + 1
  }

  RESULTIS 0
}
```

9.4 Fridays

The following program prints a table of how often the 13^{th} day of the month lies on each day of the week over a 400 year period. Since there are an exact number of weeks in 4 centuries, program shows that the 13^{th} is most of a Friday!

```
GET "libhdr"

MANIFEST { mon=0; sun=6; jan=0; feb=1; dec=11 }

LET start() = VALOF
{ LET count       = TABLE 0, 0, 0, 0, 0, 0, 0
   LET daysinmonth = TABLE 31,  ?, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31
   LET days = 0

   FOR year = 1973 TO 1973+399 DO
   { daysinmonth!feb := febdays(year)
      FOR month = jan TO dec DO
      { LET day13 = (days+12) REM 7
         count!day13 := count!day13 + 1
         days := days + daysinmonth!month
      }
   }
   FOR day = mon TO sun DO
     writef("%i3 %sdays*n",
            count!day,
            select(day,
                   "Mon","Tues","Wednes","Thurs","Fri","Sat","Sun")
           )
   RESULTIS 0
}

AND febdays(year) = year REM 400 = 0 -> 29,
                    year REM 100 = 0 -> 28,
                    year REM 4   = 0 -> 29,
                    28

AND select(n, a0, a1, a2, a3, a4, a5, a6) = n!@a0
```

9.5 Lambda Evaluator

The following program is a simple parser and evaluator for lambda expressions.

```
GET "libhdr"

MANIFEST {
// selectors
H1=0; H2; H3; H4

// Expression operators and tokens
Id=1; Num; Pos; Neg; Mul; Div;Add; Sub
Eq; Cond; Lam; Ap; Y
Lparen; Rparen; Comma; Eof
}

GLOBAL {
space:200; str; strp; strt; ch; token; lexval
}

LET lookup(bv, e) = VALOF
{ WHILE e DO { IF bv=H1!e RESULTIS H2!e
               e := H3!e
             }
  writef("Undeclared name %c*n", H2!bv)
  RESULTIS 0
}

AND eval(x, e) = VALOF SWITCHON H1!x INTO
{ DEFAULT:     writef("Bad exppression, Op=%n*n", H1!x)
               RESULTIS 0
  CASE Id:     RESULTIS lookup(H2!x, e)
  CASE Num:    RESULTIS H2!x
  CASE Pos:    RESULTIS eval(H2!x, e)
  CASE Neg:    RESULTIS - eval(H2!x, e)
  CASE Add:    RESULTIS eval(H2!x, e) + eval(H3!x, e)
  CASE Sub:    RESULTIS eval(H2!x, e) - eval(H3!x, e)
  CASE Mul:    RESULTIS eval(H2!x, e) * eval(H3!x, e)
  CASE Div:    RESULTIS eval(H2!x, e) / eval(H3!x, e)
  CASE Eq:     RESULTIS eval(H2!x, e) = eval(H3!x, e)
  CASE Cond:   RESULTIS eval(H2!x, e) -> eval(H3!x, e), eval(H4!x, e)
  CASE Lam:    RESULTIS mk3(H2!x, H3!x, e)

  CASE Ap:   { LET f, a = eval(H2!x, e), eval(H3!x, e)
               LET bv, body, env = H1!f, H2!f, H3!f
               RESULTIS eval(body, mk3(bv, a, env))
             }
  CASE Y:    { LET bigf   = eval(H2!x, e)
               // bigf should be a closure whose body is an
               // abstraction eg Lf Ln n=0 -> 1, n*f(n-1)
               LET bv, body, env = H1!bigf, H2!bigf, H3!bigf
               // Make a closure with a missing environment
               LET yf  = mk3(H2!body, H3!body, ?)
               // Make a new environment including an item for bv
               LET ne  = mk3(bv, yf, env)
               H3!yf := ne // Now fill in the environment component
               RESULTIS yf // and return the closure
             }
}
```

```
// ***************  Syntax analyser ***********************

// Construct       Corresponding Tree

// a ,.., z   -->  [Id, 'a'] ,..,  [Id, 'z']
// dddd       -->  [Num, dddd]
// x y        -->  [Ap, x, y]
// Y x        -->  [Y, x]
// x * y      -->  [Times, x, y]
// x / y      -->  [Div, x, y]
// x + y      -->  [Plus, x, y]
// x - y      -->  [Minus, x, y]
// x = y      -->  [Eq, x, y]
// b -> x, y  -->  [Cond, b, x, y]
// Li y       -->  [Lam, i, y]

LET mk1(x) = VALOF { space := space-1; !space := x; RESULTIS space }

AND mk2(x,y) = VALOF { mk1(y); RESULTIS mk1(x)  }

AND mk3(x,y,z) = VALOF { mk2(y,z); RESULTIS mk1(x)  }

AND mk4(x,y,z,t) = VALOF { mk3(y,z,t); RESULTIS mk1(x)  }

AND rch() BE
{ ch := Eof
  IF strp>=strt RETURN
  strp := strp+1
  ch := str%strp
}

AND parse(s) = VALOF
{ str, strp, strt := s, 0, s%0
  rch()
  RESULTIS nexp(0)
}
```

```
AND lex() BE SWITCHON ch INTO
{ DEFAULT:   writef("Bad ch in lex: %c*n", ch)
  CASE Eof:  token := Eof
             RETURN
  CASE ' ':
  CASE '*n' :rch(); lex(); RETURN

  CASE 'a':CASE 'b':CASE 'c':CASE 'd':CASE 'e':
  CASE 'f':CASE 'g':CASE 'h':CASE 'i':CASE 'j':
  CASE 'k':CASE 'l':CASE 'm':CASE 'n':CASE 'o':
  CASE 'p':CASE 'q':CASE 'r':CASE 's':CASE 't':
  CASE 'u':CASE 'v':CASE 'w':CASE 'x':CASE 'y':
  CASE 'z':
             token := Id; lexval := ch; rch(); RETURN

  CASE '0':CASE '1':CASE '2':CASE '3':CASE '4':
  CASE '5':CASE '6':CASE '7':CASE '8':CASE '9':
             token, lexval := Num, 0
             WHILE '0'<=ch<='9' DO
             { lexval := 10*lexval + ch - '0'
               rch()
             }
             RETURN

  CASE '-':  rch()
             IF ch='>' DO { token := Cond; rch(); RETURN }
             token := Sub
             RETURN
  CASE '+':  token := Add;    rch(); RETURN
  CASE '(':  token := Lparen; rch(); RETURN
  CASE ')':  token := Rparen; rch(); RETURN
  CASE '**': token := Mul;    rch(); RETURN
  CASE '/':  token := Div;    rch(); RETURN
  CASE 'L':  token := Lam;    rch(); RETURN
  CASE 'Y':  token := Y;      rch(); RETURN
  CASE '=':  token := Eq;     rch(); RETURN
  CASE ',':  token := Comma;  rch(); RETURN
}
```

```
AND prim() = VALOF
{ LET a = TABLE Num, 0
  SWITCHON token INTO
  { DEFAULT:     writef("Bad expression*n");    ENDCASE
    CASE Id:     a := mk2(Id, lexval);          ENDCASE
    CASE Num:    a := mk2(Num, lexval);         ENDCASE
    CASE Y:      RESULTIS mk2(Y, nexp(6))
    CASE Lam:    lex()
                 UNLESS token=Id DO writes("Id expected*n")
                 a := lexval
                 RESULTIS mk3(Lam, a, nexp(0))
    CASE Lparen: a := nexp(0)
                 UNLESS token=Rparen DO writef("')' expected*n")
                 lex()
                 RESULTIS a
    CASE Add:    RESULTIS mk2(Pos, nexp(3))
    CASE Sub:    RESULTIS mk2(Neg, nexp(3))
  }
  lex()
  RESULTIS a
}

AND nexp(n) = VALOF { lex(); RESULTIS exp(n) }

AND exp(n) = VALOF
{ LET a, b = prim(), ?

  { SWITCHON token INTO
    { DEFAULT:  BREAK
      CASE Lparen:
      CASE Num:
      CASE Id:   UNLESS n<6 BREAK
                 a := mk3(Ap,  a, exp(6));  LOOP
      CASE Mul:  UNLESS n<5 BREAK
                 a := mk3(Mul, a, nexp(5)); LOOP
      CASE Div:  UNLESS n<5 BREAK
                 a := mk3(Div, a, nexp(5)); LOOP
      CASE Add:  UNLESS n<4 BREAK
                 a := mk3(Add, a, nexp(4)); LOOP
      CASE Sub:  UNLESS n<4 BREAK
                 a := mk3(Sub, a, nexp(4)); LOOP
      CASE Eq:   UNLESS n<3 BREAK
                 a := mk3(Eq,  a, nexp(3)); LOOP
      CASE Cond: UNLESS n<1 BREAK
                 b := nexp(0)
                 UNLESS token=Comma DO writes("Comma expected*n")
                 a := mk4(Cond, a, b, nexp(0)); LOOP
    }
  } REPEAT
  RESULTIS a
}
```

```
AND try(expr) BE
{ LET v = VEC 2000
  space := v+2000
  writef("Trying %s*n", expr)
  writef("Answer: %n*n", eval(parse(expr), 0))
}

AND start() = VALOF
{ try("(Lx x+1) 2")
  try("(Lx x) (Ly y) 99")
  try("(Ls Lk s k k) (Lf Lg Lx  f x (g x)) (Lx Ly x) (Lx x) 1234")
  try("(Y (Lf Ln n=0->1,n**f(n-1))) 5")
  RESULTIS 0
}
```

9.6 Fast Fourier Transform

The following program is a simple demonstration of the algorithm for the fast fourier transform. Instead of using complex numbers, it uses integer arithmetic modulo 65537 with an appropriate N^{th} root of unity.

```
GET "libhdr"

MANIFEST {
modulus = #x10001  // 2**16 + 1

$$ln10  // Set condition compilation flag to select data size
//$$walsh

$<ln16 omega = #x00003; ln = 16 $>ln16  // omega**(2**16) = 1
$<ln12 omega = #x0ADF3; ln = 12 $>ln12  // omega**(2**12) = 1
$<ln10 omega = #x096ED; ln = 10 $>ln10  // omega**(2**10) = 1
$<ln4  omega = #x08000; ln = 4  $>ln4   // omega**(2**4)  = 1
$<ln3  omega = #x0FFF1; ln = 3  $>ln3   // omega**(2**3)  = 1

$<walsh  omega=1   $>walsh    // The Walsh transform

N       = 1<<ln    // N is a power of 2
upb     = N-1
}

STATIC   { data=0  }
```

```
LET start() = VALOF
{  writef("fft with N = %n and omega = %n modulus = %n*n*n",
                    N,         omega,     modulus)

   data := getvec(upb)

   UNLESS omega=1 DO   // Unless doing Walsh tranform
     check(omega, N)   //   check that omega and N are consistent

   FOR i = 0 TO upb DO data!i := i
   pr(data, 7)
// prints  -- Original data
//     0     1     2     3     4     5     6     7

   fft(data, ln, omega)
   pr(data, 7)
// prints   -- Transformed data
// 65017 26645 38448 37467 30114 19936 15550 42679

   fft(data, ln, ovr(1,omega))
   FOR i = 0 TO upb DO data!i := ovr(data!i, N)
   pr(data, 7)
// prints  -- Restored data
//     0     1     2     3     4     5     6     7
   RESULTIS 0
}

AND fft(v, ln, w) BE  // ln = log2 n    w = nth root of unity
{ LET n = 1<<ln
  LET vn = v+n
  LET n2 = n>>1

  // First do the perfect shuffle
  reorder(v, n)

  // Then do all the butterfly operations
  FOR s = 1 TO ln DO
  { LET m = 1<<s
    LET m2 = m>>1
    LET wk, wkfac = 1, w
    FOR i = s+1 TO ln DO wkfac := mul(wkfac, wkfac)
    FOR j = 0 TO m2-1 DO
    { LET p = v+j
      WHILE p<vn DO { butterfly(p, p+m2, wk); p := p+m }
      wk := mul(wk, wkfac)
    }
  }
}

AND butterfly(p, q, wk) BE { LET a, b = !p, mul(!q, wk)
                             !p, !q := add(a, b), sub(a, b)
                           }
```

```
AND reorder(v, n) BE
{ LET j = 0
  FOR i = 0 TO n-2 DO
  { LET k = n>>1
    // j is i with its bits is reverse order
    IF i<j DO { LET t = v!j; v!j := v!i; v!i := t }
    // k  =  100..00       10..0000..00
    // j  =  0xx..xx       11..10xx..xx
    // j' =  1xx..xx       00..01xx..xx
    // k' =  100..00       00..0100..00
    WHILE k<=j DO { j := j-k; k := k>>1 } //) "increment" j
    j := j+k                              //)
  }
}

AND check(w, n) BE
{ // Check that w is a principal nth root of unity
  LET x = 1
  FOR i = 1 TO n-1 DO { x := mul(x, w)
                        IF x=1 DO writef("omega****%n = 1*n", i)
                      }
  UNLESS mul(x, w)=1 DO writef("Bad omega**%n should be  1*n", n)
}

AND pr(v, max) BE
{ FOR i = 0 TO max DO { writef("%I5 ", v!i)
                        IF i REM 8 = 7 DO newline()
                      }
  newline()
}

AND dv(a, m, b, n) = a=1 -> m,
                     a=0 -> m-n,
                     a<b -> dv(a, m, b REM a, m*(b/a)+n),
                     dv(a REM b, m+n*(a/b), b, n)


AND inv(x) = dv(x, 1, modulus-x, 1)

AND add(x, y) = VALOF
{ LET a = x+y
  IF a<modulus RESULTIS a
  RESULTIS a-modulus
}

AND sub(x, y) = add(x, neg(y))

AND neg(x)    = modulus-x

AND mul(x, y) = x=0 -> 0,
                (x&1)=0 -> mul(x>>1, add(y,y)),
                add(y, mul(x>>1, add(y,y)))

AND ovr(x, y) = mul(x, inv(y))
```

Bibliography

[D.T64] D.T. Ross et al. AED-0 programmer's guide and user kit. Technical report, Electronic Systems Laboratory M.I.T, 1964.

[JR83] C. Jobson and J.M. Richards. *BCPL for the BBC Microcomputer*. Acornsoft Ltd, Cambridge, 1983.

[Ric] M. Richards. *My WWW Home Page*. www.cl.cam.ac.uk/users/mr/.

[Ric66] M. Richards. *The Implementation of CPL-like programming languages*. Phd thesis, Cambridge University, 1966.

[Str65] Christopher Strachey. A General Purpose Macrogenerator. *Computer Journal*, 8(3):225–241, 1965.

Appendix A

BCPL Syntax Diagrams

The syntax of BCPL is specified using the transition diagrams given in figures A.1, A.2, A.3 and A.4. Within the diagrams the syntactic categories *program*, *section*, *declaration*, *command* and $expression_n$ are represented by the rounded boxes: ─(program)─, ─(section)─, ─(D)─, ─(C)─ and ─(En)─, respectively.

The rectangular boxes are called test boxes and can only be traversed if the condition labelling the box matches the current input. When the label is a token, as in ─[WHILE]─ and ─[:=]─, it must match the next input token for the test to succeed. The test box ─[eof]─ is only satisfied if the end of file has been reached. Sometimes the test box contains a side condition, as in ─[REM n<6]─, in which case the side condition must also be satisfied. The only other test boxes are ─[is call]─ and ─[is name]─ which are only satified if the most recently read expression is syntactically a function call or a name, respectively. By setting n successively from 0 to 8 in the definition of the category ─(En)─, we obtain the definitions of ─(E0)─ to ─(E8)─. Starting from the definition of ─(program)─, we can construct an infinite transition diagram containing only test boxes by simply replacing all rounded boxes by their definitions, recursively. The parsing algorithm searches through this infinite diagram for a path with the same sequence of tokens as the program being parsed. In order to eliminate ambiguities, the left hand branch at a branch point is tried first. Notice how this rule causes the command

```
IF i>10 DO i := i/2 REPEATUNTIL i<5
```

to be equivalent to

```
IF i>10 DO { i := i/2 REPEATUNTIL i<5 }
```

and not

```
{ IF i>10 DO i := i/2 } REPEATUNTIL i<5
```

A useful property of these diagrams is that, once a test box has been successfully traversed, previous branching decisions need not be reconsidered and so the parser need never backtrack.

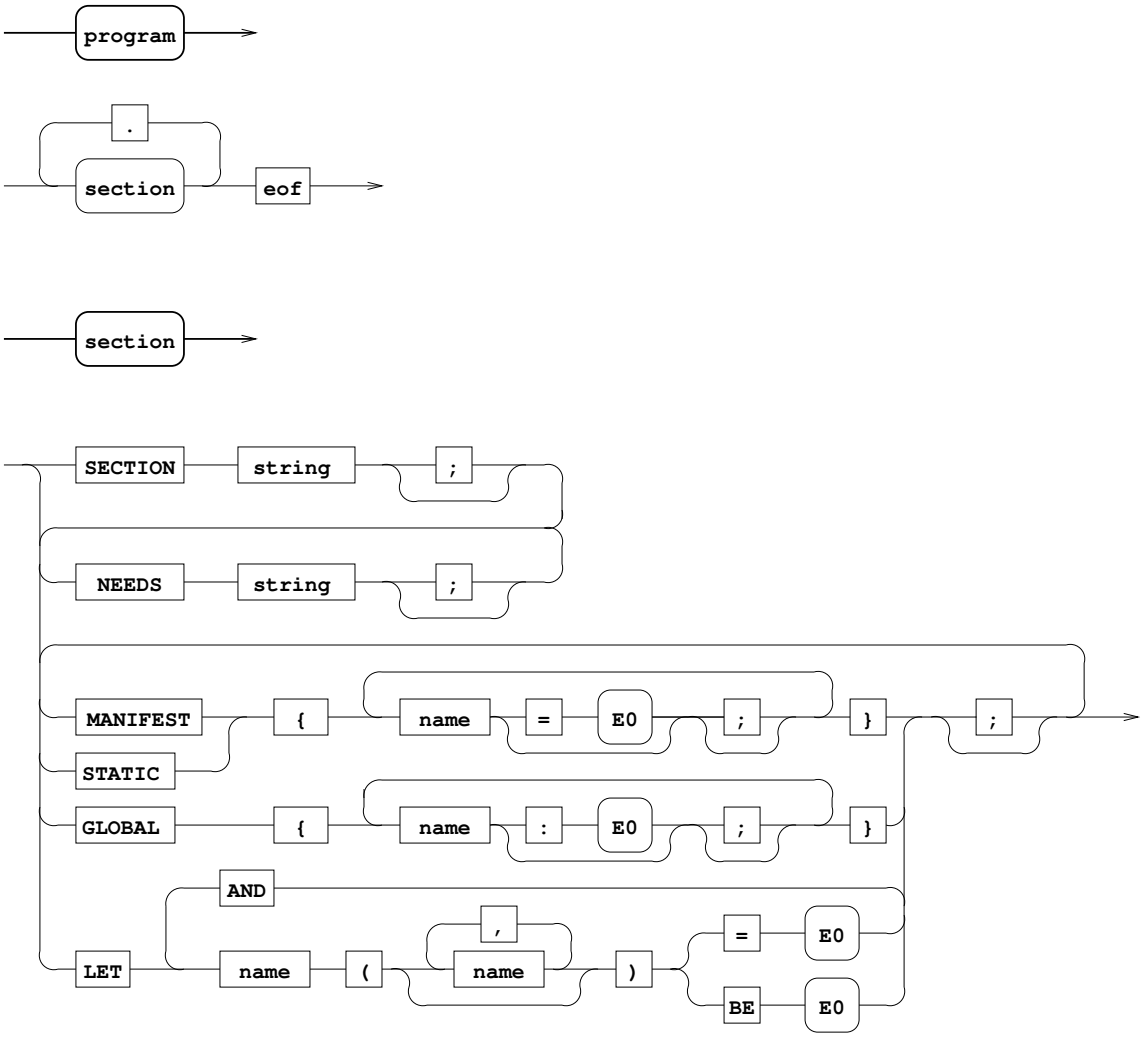


Figure A.1: Program, Section

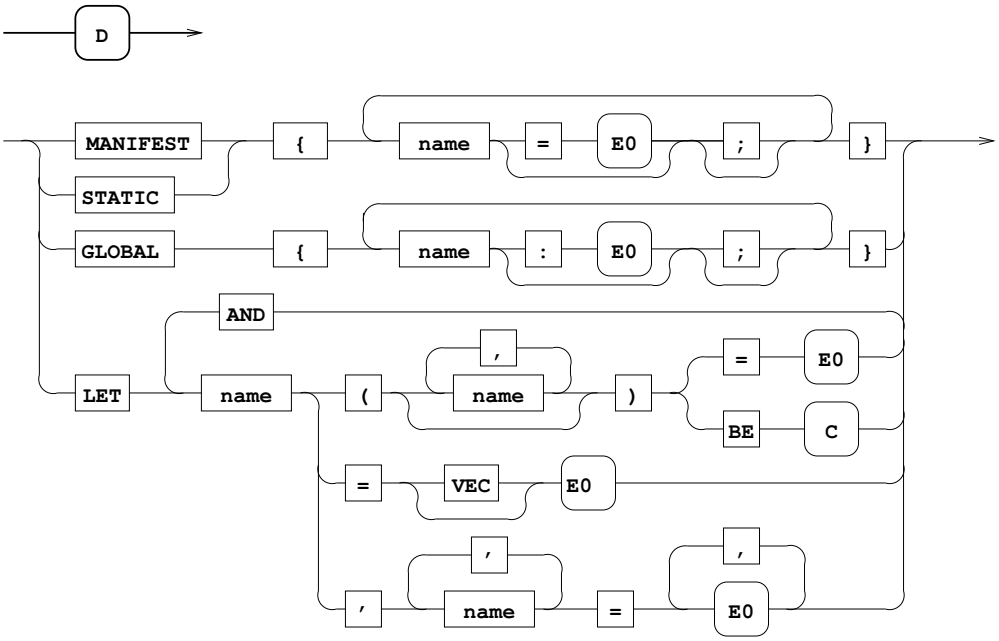


Figure A.2: Declarations

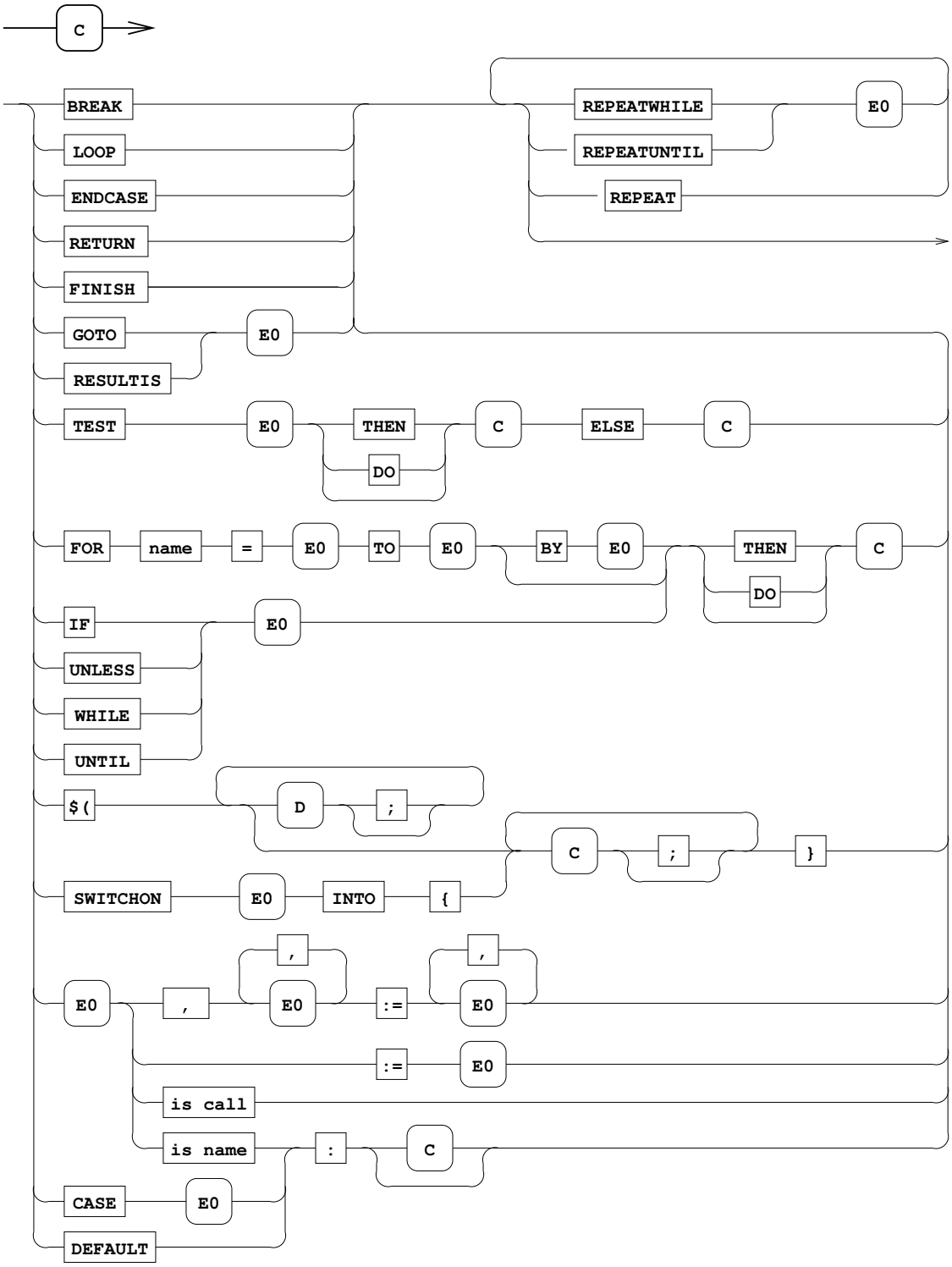


Figure A.3: Commands

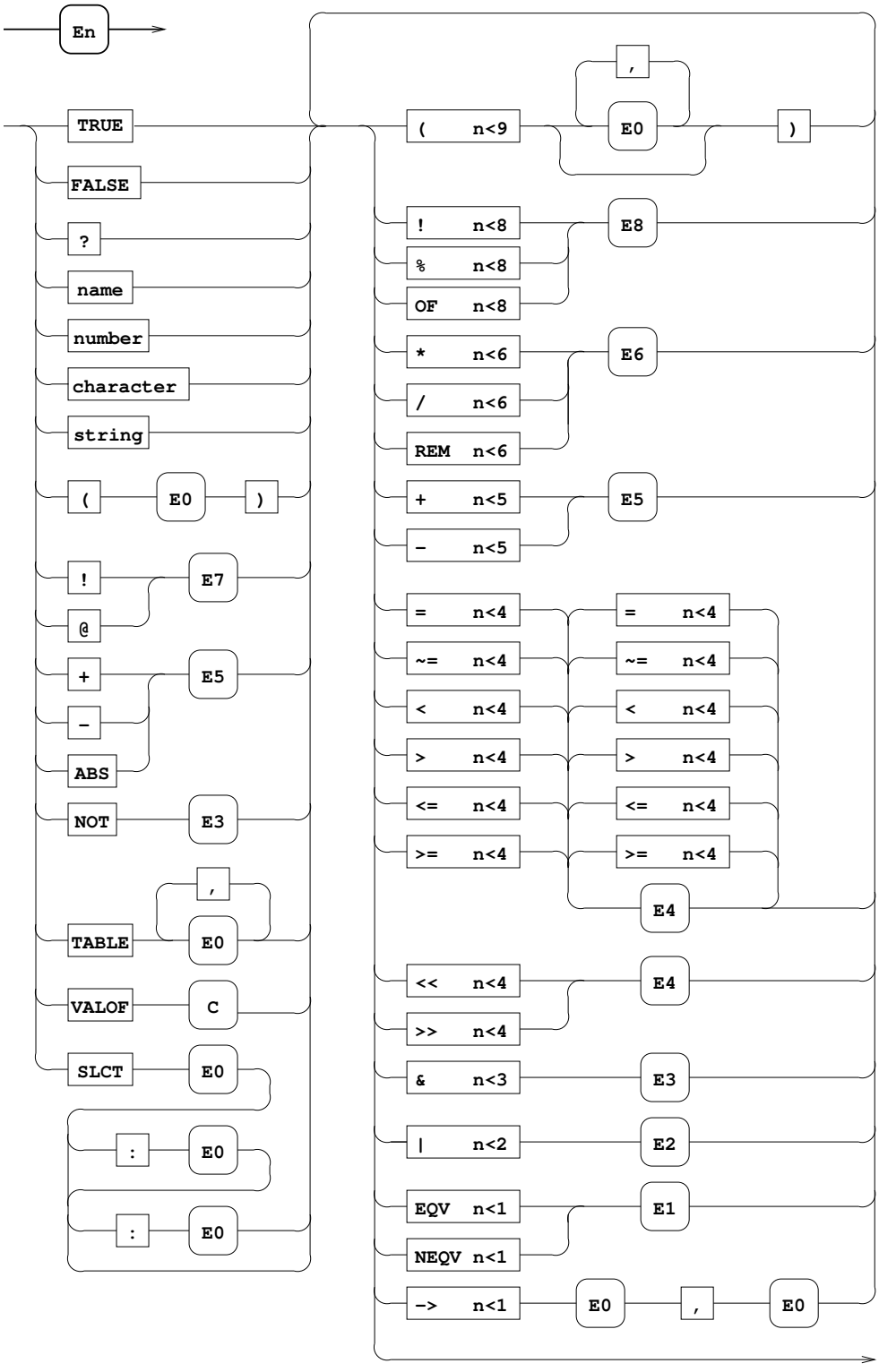


Figure A.4: Expressions