

The MCPL Programming Manual and User Guide

by

Martin Richards

`mr@uk.ac.cam.cl`

<http://www.cl.cam.ac.uk/users/mr/>

Computer Laboratory
University of Cambridge

May 23, 2007

Abstract

MCPL is a programming language that has been derived from BCPL by the inclusion of features found in ML, C and Prolog. Like BCPL, it is typeless, uses a contiguous runtime stack and has no builtin garbage collector, but it does make extensive use of ML-like pattern matching. The low level aspects of the language resemble those of BCPL and C. MCPL uses its own function calling sequence, however it is designed to allow MCPL and C functions to call each other.

Notable features of MCPL are its pattern matching facilities and the simple way in which data structures are handled.

This document gives the definition of the language, its library and how to obtain and install the system.

Keywords

Systems programming language, Typeless language, MCPL, BCPL, Mintcode, Coroutines.

Contents

1	Introduction	1
2	Language Overview	2
2.1	Comments	3
3	Expressions	3
3.1	Names and Constants	3
3.2	Vectors and Tables	4
3.3	Function Calls	5
3.4	Method Calls	6
3.5	Postfixed Expression Operators	6
3.6	Prefixed Expression Operators	7
3.7	Infix Expression Operators	7
3.8	VALOF Expressions	8
3.9	MATCH and EVERY Expressions	8
3.10	Expression Precedence	9
3.11	Manifest Constant Expressions	10
3.12	Static Constant Expressions	10
4	Commands	10
4.1	Assignments	11
4.2	Conditional Commands	12
4.3	Repetitive Commands	12
4.4	Flow of Control	13
4.5	Exception Handling	14
4.6	Sequences and Compound Commands	15
5	Patterns	16
6	Declarations	17
6.1	External Declarations	17
6.2	Manifest Declarations	18
6.3	Global Declarations	18
6.4	Static Declarations	19
6.5	Function Definitions	19
7	Scope and Extent Rules	20
8	Modules and the Interface with C	21

9	The Library	22
9.1	MSYSLIB	22
9.1.1	sys	22
9.1.2	Interpreter Management	22
9.1.3	I/O sys Operations	24
9.1.4	Other sys Operations	25
9.2	MLIB	25
9.2.1	Stream Input/Output	26
9.2.2	Input Functions	28
9.2.3	Output Functions	29
9.2.4	The Filing System	30
9.2.5	Command Arguments	31
9.2.6	Program Loading and Control	33
9.2.7	Character Handling	35
9.2.8	Coroutines	36
9.2.9	Hamming's Problem	39
9.2.10	Scaled Arithmetic	41
10	The Command Language	42
10.1	Bootstrapping	42
10.2	Commands	44
11	The Debugger	45
12	Installation	49
12.1	Linux Installation	50
12.2	Installation on Other Machines	53
12.3	The Native Code Version	53
13	Example Programs	53
13.1	Coins	53
13.2	Primes	54
13.3	Queens	54
13.4	Fridays	55
13.5	Prover	55
13.6	Eval	61
13.7	Fast Fourier Transform	64
13.8	Turing	66
	Bibliography	70

1 Introduction

The concepts underlying MCPL originates from my experience of using BCPL[RWS80] for the past 30 years and from lessons learnt using and teaching languages such as ML[Pau91], C[HS87] and Prolog[CM81] . Indeed, the name MCPL incorporates the letters ML, C and P in recognition of these languages. It is designed to be easy to learn and use, and easy to combine with C programs. Its simplicity results from its typelessness causing all expressions to yield values of the natural word size used in the implementation. This is usually 32 bits, but on some machines a 64-bit word length could be used. Even though the language is typeless, convenient access to packed 8-bit characters is provided.

A prototype interpretive implementation of MCPL is freely available via my World Wide Web home page: <http://www.cl.cam.ac.uk/users/mr/> and brief notes on how to install this distribution is given in Section 12. A native code version of the system is also available, see Section 12.3. As an introductory example, consider the following program:

```
GET "mcpl.h"

FUN start
: => writef "Enter three lengths: "
      LET a=readn(), b=readn(), c=readn()
      writef("\nTriangle entered: %d %d %d", a, b, c)
      writef("\nThis is %s triangle\n",
             sort_of_triangle(a,b,c))
.

FUN sort_of_triangle
: a, b(<a), c      => sort_of_triangle(b, a, c)
: a, b,          c(<b) => sort_of_triangle(a, c, b)

// At this point we know that a <= b <= c
: a, b,          >a+b => "not a"
: a, ?,          =a  => "an equalateral"
: a, =a,         ?   => "an isosceles"
: ?, =c,         c   => "an isosceles"
: a, b,         c   => c*c=a*a+b*b -> "a right angled",
                                     "a scalene"
.
```

The directive `GET "mcpl.h"` is like the C `#include` directive and causes a file of standard declarations to be included in the program. Function definitions are introduced by the word `FUN` followed by the function name and a list of pattern match items that test and decompose the function arguments. The function `start` has an empty pattern indicating that it is parameterless, and, as can be seen, it will output a message, read three numbers and then output two messages about the triangle whose side lengths have been entered.

The function `sort_of_triangle` returns a string that depends on its three arguments. If the second argument is less than the first, the first match item succeeds and causes `sort_of_triangle` to be re-entered with these two arguments swapped. The second match performs a similar test on the second and third arguments, and so, by the time the third match item is reached the arguments `a`, `b` and `c` are in sorted order. A compiler could notice that both calls are tail recursive, and optimise them accordingly. Indeed, the first call would be further optimised to jump to the second match item since after swapping `a` and `b` the first match cannot succeed.

The absence of types avoids the need to clutter programs and forces programmers to represent data simply. My experience is that, for many applications, data can be represented satisfactorily by means of integers, strings, bit patterns and pointers to vectors composed of these kinds. Although some say that lack of compile time type checking makes programs difficult to debug, I have not found this to be the case in practice. Indeed, there are situations where the absence of types is an advantage since it often allows a more concise and readable representation of an algorithm.

2 Language Overview

An MCPL program is made up of one or more separately compiled modules, each consisting of a list of declarations that define the constants, static data and functions belonging to the module. 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 MCPL are: expressions, commands, patterns 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. However, there is overlap between the two, since expressions can have side effects, and some commands can have results.

2.1 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 `*/`. Some care is needed when commenting out fragments of program containing string constants. Comments may not occur in the middle of multi-character symbols such as identifiers or constants.

3 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 3.10.

3.1 Names and Constants

Syntactically a name is of a sequence of letters, digits, underlines and primes(') starting with a letter. If the name starts with a capital letter, it denotes a constant that must have been declared within a **MANIFEST** declaration. If it starts with a lower case letter then it corresponds to a local variable, a static variable, a function or an external name, depending on how it was declared. Since the language is typeless, the value of a name is a bit pattern whose interpretation depends on context.

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. For instance, the following are 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 MCPL representations of the two truth values. Whenever a boolean test is made, this is performed by a comparison with **FALSE** (`=0`).

Numbers may also be represented by character constants, which consist of a single quote (') followed by zero or more characters, followed by a second single

quote. The characters are packed into 8 bit bytes to form the value, padded on the left with zeroes, if necessary. The rightmost character of the constant is the least significant byte of the result. The normal ASCII character set is used augmented by escape sequences as follows:

Escape	Replacement
<code>\n</code>	A newline (end-of-line) character.
<code>\c, \r</code>	A carriage return character.
<code>\p, \f</code>	A newpage (form-feed) character.
<code>\s</code>	A space character.
<code>\b</code>	A backspace character.
<code>\t</code>	A tab character.
<code>\e</code>	An escape character.
<code>\^c</code>	The control character <i>c</i> , for any appropriate <i>c</i> .
<code>\"</code>	"
<code>\'</code>	'
<code>\\</code>	\
<code>\ddd</code>	The single character with number <i>ddd</i> (one or more decimal digits denoting an integer in the interval [0,255]).
<code>\f..f\</code>	This sequence is ignored, where <i>f..f</i> 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 to its first byte in memory. For compatibility with C, a zero byte is appended to mark the end of the string.

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

```
sendpkt(P_notinuse, rdtask, ?, ?, Read, buf, size)
LET op=token, a=1, b=?, ptr=[3,?,?,?]
```

3.2 Vectors and Tables

An expression of the form: $[E_0, \dots, E_n]$ returns a pointer to $n + 1$ consecutive locations of dynamic memory initialised with the values of the expressions E_0, \dots, E_n . The space is allocated when control passes into the current dynamic

scope (see Section 7 on Scope and Extent). It is released when execution leaves the current dynamic scope. The pointer behaves like a vector with bounds 0 and n , and its elements can be accessed using the subscripting operator `!`, described Section 3.7. The initialising expressions E_0, \dots, E_n may, of course, contain dynamic vectors, tables and strings, and so an expression such as:

```
[ '->', ['=', [Id, "x"], [Numb, 0]],
          [Numb, 0],
          [Id, "abc"]
]
```

is legal. However, remember that the space for this dynamic structure only remains allocated as long as control remains within the current dynamic scope.

Uninitialised dynamic vectors of words or characters can be created by expressions of the form: `VEC K` or `CVEC K`, respectively, where K is a manifest constant expression (see Section 3.11) giving the upper bound. The lower bound is always zero. The space for a dynamic vector is only allocated while control is within the current dynamic scope, see Section 7.

A static vector can be created using an expression of the following form: `TABLE [K0, ..., Kn]` where K_0, \dots, K_n are static constant expressions, see Section 3.12. The space for a static vector is allocated for the lifetime of the program.

3.3 Function Calls

A function call is syntactically an expression followed by an argument list, as in the following examples.

```
newline()
mk3(Mult, x, y)
writef "Hello\n"
f[1,2,3]
(fntab!i)(p, [a, b])
```

Multiple arguments are enclosed in parentheses and an empty argument list is represented by `()`. A single argument only needs to be enclosed in parentheses if it is not a name, a number, a string, `TRUE`, `FALSE`, `?` or an expression starting with `[`. The last example above illustrates a call in which the function is specified by an expression.

It is possible, using an `EXTERNAL` declaration (see Section 6.1), to declare a function that uses the C calling sequence. A call that explicitly uses a function name declared in this way will be compiled using the C calling sequence. All other calls use the MCPL calling sequence. If a function is defined in the scope of an `EXTERNAL` declaration of the same name, then the function will be accessible

from other modules, and, if it was declared to use the C calling sequence, it will be callable from C directly. This scheme allows for convenient mixing of C and MCPL programs.

If a call occurs in the context of an expression then it is expected to return a result. The result, if any, of a call in any other context is thrown away.

3.4 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:

$$E\#(E_1, \dots, E_n)$$

is defined to be equivalent to:

$$(E_1!0!E)(E_1, \dots, E_n)$$

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 expression. An example program illustrating method calls can be found in `MCPL/mcplprogs/objdemo.m` in the MCPL distribution system (see Section 12).

3.5 Postfixed Expression Operators

Expressions of the form: $E++$, $E+++$, $E--$ or $E---$ cause the location specified by E to be incremented or decremented. The result of the expression is its original value, before modification. The operator $++$ increments by one, $+++$ increments by the number of bytes in a word, $--$ decrements by one and $---$ decrements by the number of bytes in a word. In addition to working with integers, the operators $++$ and $--$ adjust byte pointers to point to adjacent bytes, and $+++$ and $---$ adjust word pointers to point to adjacent words. It is thus assumed, in MCPL, that pointers to adjacent bytes are integers that differ by one. Pointers to adjacent words differ by an amount `Bpw` (declared in `mcpl.h`) is implementation dependent, but is usually 4 and sometimes 8.

3.6 Prefixed Expression Operators

Expressions of the form: $++E$, $+++E$, $--E$ or $---E$ cause the location specified by E to be incremented or decremented. The result of the expression is its value after modification. As with the post fixed operators, $++$ and $--$ increment or decrement by one, while $+++$ and $---$ increment and decrement by B_{pw} . They work as expected with pointers.

An expression of the form $!E$ returns the contents of the memory word pointed to by the value of E , and an expression of the form $\%E$ returns an unsigned integer equal to the byte pointed to by the value of E .

An expression of the form $\@E$ returns a pointer to the 8 bit or word sized memory location specified by E . E can only be a variable name or an expression with leading operator $!$ or $\%$.

Expressions of the form: $+E$, $-E$, $\sim E$, $ABS E$ or $NOT E$ 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, \sim returns the bitwise complement of the value, ABS returns the absolute value, and NOT returns the boolean complement of the value.

3.7 Infix 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 expressions of the form $E_1\ll E_2$ (or $E_1\gg E_2$) 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 , $E_1 \text{ MOD } E_2$, $E_1 \& E_2$, $E_1 \text{ XOR } E_2$, $E_1 + E_2$, $E_1 - E_2$, and $E_1 | E_2$ return the result of applying the given operator to the two operands. The operators are, respectively, integer multiplication, integer division, remainder after integer division, bitwise AND, bitwise exclusive OR, integer addition, integer subtraction, and bitwise OR.

An expression of the form: $E \text{ relop } E \text{ relop } \dots \text{ relop } E$ where each *relop* is one of $=$, $\sim =$, \leq , \geq , $<$ 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. No operand is evaluated more than once.

An expressions of the form: E_1 AND E_2 or E_1 OR E_2 returns the boolean value obtained by applying the given operator to the boolean values of E_1 and E_2 . If the result can be determined from E_1 alone, then E_2 is not evaluated.

An expression of the form: $E_1 \rightarrow E_2, E_3$ first evaluates E_1 , and, if this yields FALSE, it returns the value of E_3 , otherwise it return the value of E_2 .

3.8 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 RESULT E within C , execution terminates, returning the value of E as the result of the VALOF expression. The command C is in a new dynamic scope (see Section 7).

3.9 MATCH and EVERY Expressions

A MATCH expression has the following form:

```
MATCH args
:  $P, \dots, P \Rightarrow Clist$ 
...
:  $P, \dots, P \Rightarrow Clist$ 
.
```

It consists of the word MATCH followed by an argument list, followed by zero or more match items (described in Section 5). The list of match items must be terminated by a dot (.). The argument list is either a single expression or a list of zero or more expressions enclosed in parentheses. These arguments are evaluated and placed in consecutive stack locations before passing control to the first match item. If all the patterns of this item match successfully, control passes to its command list, otherwise control passes to the next match item. Execution of the MATCH expression is complete when execution of the selected command list finishes. If the last executed command yields a result, this is returned as the result of the MATCH expression. If no match item is successful, a PATERR exception is raised (see Section 4.5).

An EVERY expression is syntactically identical to a MATCH expression with the word MATCH replaced by the word EVERY. It has the same meaning except that, when the execution of the selected command sequence is complete, control is passed to the next match item. Thus, the command lists of all successful match items are executed. An EVERY expression does not yield a result.

3.10 Expression Precedence

A lexical token that can start an expression or a pattern cannot denote an infix or postfix operator when occurring as the first token of a line. This rule applies to the tokens: `!`, `%`, `+`, `++`, `+++`, `-`, `--`, `---`, `=`, `~=`, `<=`, list of syntactically the same as the argument list of a function call and consists of either a single argument (a name, a number, a string, `TRUE`, `FALSE`, `?` or an initialised dynamic vector), or it is a list of arguments enclosed in parentheses.

`>=`, `<` and `>`, and its purpose is to allow semicolons at the end of lines to be omitted.

Table 1 specifies the precedence of the various expression constructs. The precedence values are in the range 0 to 15, 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 `v!i!j` is equivalent to `(v!i)!j`, while `b1->x,b2->y,z` is equivalent to `b1->x,(b2->y,z)`.

15L	Names, Literals, <code>?</code> , <code>TRUE</code> , <code>FALSE</code> , (<code>E</code>), [<code>E</code> , . . . , <code>E</code>], Function and method calls	
14	<code>TABLE</code> <code>VEC</code> <code>CVEC</code>	
13	<code>++</code> <code>+++</code> <code>--</code> <code>---</code>	Postfixed
12	<code>++</code> <code>+++</code> <code>--</code> <code>---</code>	Prefixed
11L	<code>!</code> <code>%</code>	Dyadic
10	<code>!</code> <code>%</code> <code>@</code> <code>~</code> <code>+</code> <code>-</code> <code>ABS</code>	Prefixed
9L	<code><<</code> <code>>></code>	Dyadic operators
8L	<code>*</code> <code>/</code> <code>MOD</code> <code>&</code>	
7L	<code>XOR</code>	
6L	<code>+</code> <code>-</code> <code> </code>	
5	<code>=</code> <code>~=</code> <code><=</code> <code>>=</code> <code><</code> <code>></code>	Extended Relations
4	<code>NOT</code>	Truth value operators
3L	<code>AND</code>	
2L	<code>OR</code>	
1R	<code>-></code> <code>,</code>	Conditional expression
0	<code>VALOF</code> <code>MATCH</code> <code>EVERY</code>	Miscellaneous

Table 1: Operator precedence

Notice that these precedence values imply that

! f x	means	! (f x)
! p +++	means	! (p +++)
! @ x	means	! (@ x)
! v ! i ! j	means	! ((v!i)!j)
@ v ! i ! j	means	@ ((v!i)!j)
@ "abc" % i	means	@ ("abc" % i)
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)

3.11 Manifest Constant Expressions

A manifest constant expression is an expression that can be evaluated at compile time. It may only consist of manifest constant names, numbers and character constants, TRUE, FALSE, ?, the operators ~, ABS, <<, >>, &, MOD, *, /, XOR, +, -, |, the relational operators, NOT, AND, OR, and conditional expressions. Manifest expressions are used in MANIFEST declarations, FOR commands and as the operand of VEC or CVEC.

3.12 Static Constant Expressions

A static constant expression may be used to specify the initial value of a static location. It may be a string, a static vector, a function, the address of a static variable or a manifest constant expression. Within a static constant expression, the following constructs are allowed and declare static vectors: VEC K and CVEC K, where K denote a manifest constant expression, and $[K_0, \dots, K_n]$, where K_i denote static constant expressions.

4 Commands

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

4.1 Assignments

A command of the form $L:=E$ causes a location specified by the expression L to be updated by the value of expression E . Some example assignments are as follows:

```
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. External and function names do not denote updatable variables. If L has leading operator `!`, then its evaluation (given in Section 3.7) 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 simultaneous assignment has the following form:

$$L_1, \dots, L_n := E_1, \dots, E_n$$

Here the locations to update and the values of the right hand side expressions are all determined before the assignments are performed. Thus the assignment $x,y:=y,x$ will swap the values of x and y , and the assignment $i,v\%i:=i+1,ch$ will use the original value of i as the subscript of v . The assignments are however performed in undefined order and so $x,x:=1,2$ may set x to either 1 or 2.

If the same value is to be assigned to several locations then an assignment of the following form can be used.

$$L_1, \dots, L_n \text{ ALL} := E$$

An assignment of the form:

$$L_1, \dots, L_n \text{ op} := E_1, \dots, E_n$$

where $op:=$ is one of: `<<:=`, `>>:=`, `&:=`, `*:=`, `/:=`, `MOD:=`, `XOR:=`, `|:=`, `+=` or `-=` is evaluated as follows. First, the left hand side locations and the values of the right hand side expressions are determined, then the assignments are performed. Each assignment updates a location with the result obtained by applying the given operator to the previous contents of the location and the value given by the right hand side. The assignment order is undefined.

The statement:

$$\text{LET } V_1=E_1, \dots, V_n=E_n$$

declares each variable V_i to be local within the current scope and assigns to it the corresponding value E_i . The space for these variable is allocated on entry to the

current scope, but the assignment taken place when the statement is executed. Each initialisation component ($=E_i$) is optional, so, for example:

```
LET a=1, b, c=10
```

declares *a*, *b* and *c*, but only assigns values to *a* and *c*.

4.2 Conditional Commands

The syntax of the three conditional commands is as follows:

```
IF E DO C1
UNLESS E DO C2
TEST E THEN C1 ELSE C2
```

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.

4.3 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 vid = E1 TO E2 DO C
FOR vid = E1 TO E2 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 (*vid*) to the value of *E*₁, and evaluates the end limit *E*₂. Until *vid* moves beyond the end limit, the command *C* is executed and *vid* increment by the step length given by *K* which must be a manifest constant expression (see Section 3.11). If BY *K* is omitted

BY 1 is assumed. A FOR command starts a new dynamic scope (see Section 7) and the control variable *vid* is allocated a location within this new scope, as are all other dynamic variables and vectors within the FOR command.

4.4 Flow of Control

The following commands affect the flow of control.

```
RESULT  
RESULT E  
EXIT  
EXIT E  
RETURN  
RETURN E  
LOOP  
BREAK  
GOTO args
```

RESULT causes evaluation of the current VALOF expression to complete. If the expression *E* is present, its value becomes the result of the VALOF expression.

RETURN causes evaluation of the current function to terminate, returning the value of *E*, if present.

LOOP causes control to jump to the point just after the end of the body of the smallest textually enclosing repetitive command (see Section 4.3). 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 other repetitive 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.

EXIT causes evaluation of the command list of the smallest enclosing match item to complete. If the expression *E* is present, its value becomes the result if appropriate. Match items are described in Section 5 and are used in the following four match constructs: MATCH expressions, EVERY expressions, HANDLE commands and function definitions.

The GOTO command takes an argument list which is syntactically similar to the argument list of a MATCH expression (see Section 3.9). The arguments are evaluated and assigned to the argument locations belonging to the smallest enclosing match construct. Control is then passed to its first match item.

4.5 Exception Handling

An expression of the form:

`RAISE args`

causes an exception to be raised. The argument list is syntactically similar to the argument list of a `MATCH` expression, but may only contain at most three arguments. These arguments are assigned to the argument locations belonging to the currently active exception handler and then control is passed to its first match item. This transfer of control may involve returning from one or more function activations. If none of the match items of the current handler are successful, then the same arguments are passed to the handler one level further out. By convention, the first exception argument is an integer specifying the exception, with zero reserved for the match exception (`PATERR`), and other small numbered exceptions reserved for use by standard library functions.

An exception handler is declared by a construct of the following form:

```
C HANDLE : plist => clist
          ...
          : plist => clist
          .
```

It executes the command *C* in an environment in which the exceptions will be matched against the given list of match items. On completing the command, the previous exception environment is restored.

A simple example of how exceptions can be used is demonstrated by the following program.

```
GET "mcpl.h"

MANIFEST
  Id, Num, Mult, Div, Pos, Neg, Plus, Minus,
  Lookup=100, Eval

FUN lookup
: n,          0 => RAISE(Lookup, n)
: n, [=n, val, ?] => val
: n, [ ?,   ?, e] => lookup(n, e)
.

FUN eval
: [Id, x],      e => lookup(x, e)
: [Num, k],     ? => k
: [Pos, x],     e => eval(x, e)
: [Neg, x],     e => - eval(x, e)
: [Mult,x,y],  e => eval(x, e) * eval(y, e)
: [Div,x,y],   e => eval(x, e) / eval(y, e)
: [Plus,x,y],  e => eval(x, e) + eval(y, e)
: [Minus,x,y], e => eval(x, e) - eval(y, e)
: [op],        ? => RAISE (Eval, op)
.

FUN start : =>
{ LET exp = [Mult, [Num,23], [Plus,[Id,'a'],[Id,'b']]]
  LET env = ['a',36,['b',19,0]]
  writef("Result is %d\n", eval(exp, env) )
} HANDLE
: Lookup, id => writef ("Id %c not declared\n", id)
: Eval,   op => writef ("Unknown operator %d in eval\n", op)
.
.
```

4.6 Sequences and 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. A semicolon can be omitted if it is the last symbol on a line. For this to work, expression and pattern operators which can be both prefixed operators and infix or postfix may only occur as the first token on a line with its prefixed meaning (see Section 3.10). The first symbol of a function argument list must be on the same line as the last symbol of the expression specifying the function.

It is sometimes necessary to group a sequence of commands to behave syntactically as a single command. Curly brackets (`{}`) are used for this purpose.

An expression is allowed wherever a command is permitted, and is evaluated at the same time that such a command would be. This is useful for invoking functions and executing the pre- and post-incrementing and decrementing expressions.

The result of any such expression is thrown away unless it is the last position of a sequence.

Empty commands are allowed.

5 Patterns

Pattern matching is one of the most important facilities provided within MCPL since it allows both a mechanism for multiway selection and a means of associating variable names with locations in memory. Patterns are used in function definitions, the `MATCH` and `EVERY` constructs, and in the exception handling mechanism. Within each of these constructs, the user may supply a list of match items, which must be terminated by a dot (`.`). Each match item has the following syntactic form:

$$: Plist \Rightarrow Clist$$

where *Plist* is a list of zero or more patterns separated by commas and *Clist* is a sequence of commands, separated by semicolons or newlines. At the moment when control is passed to a match item, there is a list of argument values laid out in consecutive locations of memory. The patterns in the pattern list are matched against these arguments in left to right order. If all matches are successful then control passes to the corresponding command sequence.

A pattern can be an explicit numerical constant (e.g. `1234` or `'A'`) or a range (e.g. `0..9` or `'A'..'Z'`) or an alternation of constants or ranges (e.g. `2 | 3 | 5 | 10..20`). An argument matches such a pattern if

- (a) it equals the number,
- (b) it lies within the given range
- or (c) it is matched by one of the alternations.

Manifest names may be used within patterns wherever numbers are allowed. The patterns `TRUE` and `FALSE` match the corresponding values in the current argument, and a question mark (`?`) or empty pattern will match any argument value.

A pattern consisting of a variable name always matches successfully and is treated as a declaration of the name, causing it to be attached to the corresponding argument location. The scope of the name is the entire pattern list and associated command sequence.

An argument which is a pointer can be matched by a pattern of the following form:

$$[Plist]$$

where *Plist* is a list of patterns separated by commas. If this pattern occurs, the corresponding argument is assumed to be a valid pointer into memory. The consecutive locations pointed to are matched in turn by the patterns in *Plist*. This construct can be nested to any depth.

A pattern may consist of a relational operator (=, ~=, <=, >=, < or >) followed by an expression. The corresponding argument is compared with the value of the expression to determine the success of the match.

A pattern consisting of one of the assignment operator (:=, <<:=, >>:=, &:=, *:=, /:=, MOD:=, XOR:=, |=, += or -=) followed by an expression is always successful. It has the side effect that, if the entire pattern matches successfully, the assignment is performed with the implied left hand operand, just prior to executing the corresponding command sequence.

If two patterns are juxtaposed, then both are matched against the same argument location. For instance, in

```
      :  sum, coins[val(<=sum)] => ...
```

`coins` matches the second argument, which must be a pointer to a memory location which is given the name `val` and which must be less than or equal to the first argument (`sum`).

6 Declarations

Declarations are used to declare external names, manifest constants, global variables, static variables and to define functions, and they may only occur at the outermost level of a program. It is thus not possible to declare a function, for instance, within another function. The five kinds of declaration are described below.

6.1 External Declarations

An external declaration consists of the word **EXTERNAL** followed by a list of names that are possibly qualified by type information. All external names start with lower case letters. The following example declaration:

```
EXTERNAL muldiv, printf/vpi_, calloc/puu
```

declares the external names `muldiv`, `printf` and `calloc`. Since `muldiv` is unqualified, it is declared to be an external MCPL function, while the qualified names `printf` and `calloc` are taken to be external C functions. A qualifier consists of a slash (/) followed by a sequence of lower case letters possibly terminated by an underline (_), and it specifies the type of the external C function. Each letter denotes a data type, as follows:

v	The type <code>void</code> .
p	A pointer type.
s	The type <code>short</code> .
i	The type <code>int</code> .
u	The type <code>unsigned int</code> .
l	The type <code>long int</code> .
-	This can occur as the last character of a qualifier to indicate that the function takes a variable number of arguments, compatible with the <code>varargs</code> mechanism in C.

The first letter of a qualifier specifies the result type of the function and the remaining letters specify the types of the first few arguments. If an MCPL call of a C function supplies more arguments than the specification declares then the extra arguments are assumed to be of type `long int`.

If an MCPL function is declared in the scope of an external declaration for the same name then the function is declared as an external entry point and can be called from other modules. If the external declaration specifies a C function type, then the MCPL function is compiled to be compatible with the C calling sequence.

6.2 Manifest Declarations

A `MANIFEST` declaration has the following form:

```
MANIFEST  $N_1=K_1, \dots, N_n=K_n$ 
```

where N_1, \dots, N_n are constant identifiers (see Section 3.1) and K_1, \dots, K_n are manifest constant expressions (see Section 3.11). It may only occur at the outermost level of a program. Each name is declared to have the constant value specified by the corresponding manifest expression. If a value specification ($=K_i$) is omitted, then 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.

6.3 Global Declarations

A global declaration consists of the word `GLOBAL` followed by a list of names, each possibly followed by a position specifier $:K$ where K is a manifest expression (see Section 3.11). For example, the following declaration:

```
GLOBAL a, b:200, c, d:251
```

declares the global variables `a`, `b`, `c` and `d`. They are elements of a globally accessible vector. The position a variable is given by the position specifier if provided, otherwise it is the position one greater than occupied by the previous variable. If the first variable in a global declaration list has no specifier then it is given position zero by default. Thus the global variables `a`, `b`, `c` and `d` declared above occupy positions 0, 200, 201 and 251, respectively.

6.4 Static Declarations

A static declaration consists of the word `STATIC` followed by a list of names, each possibly initialised by a static constant expressions (see Section 3.12). For example, the following declaration:

```
STATIC a=541, b, c=[123,"Hi"]
```

declares the static variables `a`, `b` and `c`. The initial value given to `a` is 541. The initial value of `b` is, by default, 0 and the initial value of `c` is the value of the static expression `[123,"Hi"]`, that is, a vector of two elements initialised with the integer 123 and a pointer to the string "Hi".

6.5 Function Definitions

A function definition has the following form:

```
FUN name : P , ... , P => Clist
      .
      .
      .
      : P , ... , P => Clist
      .
```

It consists of the word `FUN` followed by the name of the function (which must start with a lower case letter), followed by zero or more match items (described in Section 5) having the same syntax as in `MATCH` expressions (see Section 3.9). When a function is called, the arguments are placed in consecutive stack locations and control is passed to the first match item. If all the patterns of this item successfully match their corresponding arguments then control passes to its command list, otherwise control passes to the next match item. After executing the selected command list, control returns back to the call. If the last executed command yields a result then this is returned as the result of the function.

If no match items are successful, a `PATERR` exception is raised (see Section 4.5). Notice that functions defined without match items automatically generate a `PATERR` exceptions when called. This feature is useful during program development.

Since functions have no dynamic free variables, the calling overhead is small. It is permissible to pass functions as arguments to other functions, assign them to variables or return them as function results. They are also allowed in static constant expressions.

If a function declaration occurs in the scope of a global declaration of the same name, then it is the initial value of the corresponding global variable. This provides a means of calling functions defined in separately compiled modules without using external declarations.

7 Scope and Extent Rules

All identifiers used in a program must have a declaration and must be used only within the scope of that declaration. The scope of an identifier is the textual region of program for its declaration is valid, and this depends on the kind of declaration.

MANIFEST identifiers are syntactically distinct from all other identifiers and can be considered separately. **MANIFEST** declarations are treated as if they were all brought to the head of the entire program, and so the scope of a **MANIFEST** identifier is the entire program. It is permissible to use a manifest identifier in the same **MANIFEST** declaration provided its use occurs later in the declaration list.

Identifiers declared by **EXTERNAL** declarations are regarded as being declared at the start of the program, and have a scope that extends throughout the entire program. All identifiers declared by function declarations are regarded as declared immediately after the externals and so have a similar scope. All identifiers declared by **GLOBAL** declarations are regarded as declared immediately after the function identifier. **STATIC** identifiers are regarded as being declared next.

The function definitions occur within the scope of all these declarations.

All other identifiers are declared within functions. These are called dynamic variables since they come into existence when a function is called and cease to exist when the function returns. Multiple activations of such variables can occur if a function is called recursively.

The scope of a dynamic variable is the smallest *dynamic scope* containing its declaration. A dynamic scope is defined to be any region of program that

1. starts at the colon (:) of a match item and extends to the end of its command sequence,
2. starts just after the symbol => of a match item and extends to the end of the command sequence,

3. starts just after the symbol { and extends to the matching },
4. is the body of a VALOF expression, or
5. extends from the word FOR to then end of the body of the FOR command.

Thus, the scope of an identifier declared within a pattern extends forwards and backwards to include the entire pattern and command sequence of the current match item, and the scope of identifiers declared by a LET declaration extends forwards and backwards to include the whole of the smallest enclosing dynamic scope, which usually starts at the nearest preceding =>, VALOF, FOR or { symbol.

Dynamic scopes also determine the dynamic lifetime (or extent) of function arguments, variables declared by LET and local vectors declared by VEC, CVEC or [E, . . . , E]. Space for such variables and vectors is allocated when control passes into the current dynamic scope, and is freed when control leaves this scope.

8 Modules and the Interface with C

An MCPL program may start with a declaration of the following form:

```
MODULE name
```

where `name` is a variable name starting with a lower case letter. This declares `name` to be an externally accessible pointer to the start of the loaded module. This pointer may be used to gain access to some implementation dependent data held at the start of the module, such as its size, its name, its date of compilation etc.

If an MCPL program calls a function whose name was declared by an `EXTERNAL` declaration to be a C function, then the C calling sequence is used, passing the arguments as prescribed by the `EXTERNAL` declaration. If a function is declared in the scope of an `EXTERNAL` declaration then it can be called from other modules. If the `EXTERNAL` declaration declared it to be a C function, it is compiled to be compatible with the C calling sequence. `EXTERNAL` names that have not been specified to be C functions have values that are either entry points to MCPL functions (possibly in other modules) or pointers to external static variables. Access to the values of such a variable requires an indirection (using either ! or %).

9 The Library

This and the following sections describe the standard library functions for the interpretive MCPL system. The interpreter is called `minterp` and is implemented in C. Sometimes the system contains a second faster interpreter called `mintasm` that is implemented in assembly language and provided fewer debugging aids.

The library functions are defined in `MSYSLIB` and `MLIB` and use the first 200 elements of the global vector. 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.

9.1 MSYSLIB

This module contains the definitions of the functions `sys`, `chgco` and `muldiv`. These functions are hand written in Mintcode because they need to execute the Mintcode instructions (`SYS`, `CHGCO` and `MDIV`) that cannot be generated by the MCPL compiler. The instruction `SYS` provides an interface with the host operating system, `CHGCO` is used in the implementation of coroutines (see Section 9.2.8), and `MDIV` performs the operation required by `muldiv`(see Section 9.2.10).

9.1.1 `sys`

The Mintcode instruction `SYS` causes the interpreter to call the C function `dosys`. The current stack frame and global vector pointers are passed as arguments of `dosys` so that it can access any local or global variables belonging to the interpreted code. The `SYS` instruction is only invoked by the MCPL function `sys` that is defined in `MSYSLIB`. A typical call of `sys` is as follows:

$$res := sys(op, \dots)$$

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.

9.1.2 Interpreter Management

`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 MCPL Mintcode System to terminate. If `code` equal `-1`, interpretation continues using the fast interpreter (`mintasm`). If `code` equal `-2`, interpretation continues using the slow interpreter (`minterp`).

`res := sys(1, regs)`

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

<code>regs!0</code>	A register	– work register
<code>regs!1</code>	B register	– work register
<code>regs!2</code>	C register	– work register
<code>regs!3</code>	P register	– the stack frame pointer
<code>regs!4</code>	G register	– the base of the global vector
<code>regs!5</code>	ST register	– the status register (unused)
<code>regs!6</code>	PC register	– the program counter
<code>regs!7</code>	Count register	– see below
<code>regs!8</code>	H register	– used for exception handling

The count register is decremented every time an instruction is interpreted, and the interpreter returns when the count reaches zero. A count of `-1` is treated as infinity and may cause a faster interpreter `mintasm` (possibly implemented in assembly language) to be used.

The interpreter returns when it encounters either a fault or a call of `sys(0, ...)`. On return, the Mintcode registers is saved back into `regs` and the returned result is the return code. The builtin return codes are as follows:

<code>-2</code>	Use the slow interpreter <code>minterp</code>
<code>-1</code>	Use the fast interpreter <code>mintasm</code>
<code>0</code>	Normal successful completion (by convention)
<code>1</code>	Non existant Cintcode instruction
<code>2</code>	BRK instruction encountered
<code>3</code>	Count has become negative
<code>4</code>	PC set to a negative value
<code>5</code>	Division by zero

`sys(2)` – Turn on tracing of the Mintcode instruction execution.

`sys(3)` – Turn off tracing (`minterp` only).

`sys(4)` – Clear the tally vector and start tallying (`minterp` only).

`sys(5)` – Stop tallying (`minterp` only).

The calls `sys(2)` and `sys(3)` provide a primitive trace facility to aid debugging the interpreter itself and are not normally of use to ordinary users. 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 Mintcode instructions executed. When tallying is enabled, the i^{th} element of the tally vector is incremented by one every time the instruction at location i of the Mintcode memory is executed. The upper bound 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`.

9.1.3 I/O sys Operations

`ch := sys(10)`

Return the next character from the keyboard.

`sys(11, ch)`

Send character `ch` to the standard output (normally the screen).

`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.

`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.

`sys(16, fp)`

This closes the file whose file pointer is `fp`.

9.1.4 Other sys Operations

The remaining `sys` operations are as follows:

```

sys(17, name)      FUN deletefile : name      => sys(17, name).
sys(18,old,new)    FUN renamefile : old, new => sys(18, old, new).
sys(21, upb)      FUN getvec    : upb      => sys(21, upb).
sys(22, ptr)      FUN freevec   : ptr      => sys(22, ptr).
sys(23, name)     FUN loadseg   : name     => sys(23, name).
sys(24, seg)      FUN globin    : seg      => sys(24, seg).
sys(25, seg)      FUN unloadseg : seg      => sys(25, seg).
sys(26,a,b,c)     FUN muldiv    : a, b, c  => sys(26,a,b,c).
sys(27,n,val)     FUN setraster : n, val  => sys(27,n,val).
sys(28)           FUN intflags  :          => sys(28).

```

These are described later.

9.2 MLIB

This section describes the MLIB library function and their associated data structures.

When the Mintcode System is started, a region of store is allocated for the Mintcode memory. It is where Mintode 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 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
<code>rootnode!Rtn_membase</code>	Pointer to the start of the Mintcode memory.
<code>rootnode!Rtn_memsize</code>	The size of the Mintcode memory in words.
<code>rootnode!Rtn_blklist</code>	The start of the chain of memory blocks.
<code>rootnode!Rtn_tallyv</code>	The tally vector.
<code>rootnode!Rtn_syslib</code>	The MSYSLIB code segment.
<code>rootnode!Rtn_blib</code>	The MLIB code segment.
<code>rootnode!Rtn_boot</code>	The MBOOT code segment.
<code>rootnode!Rtn_cli</code>	The MCLI code segment.
<code>rootnode!Rtn_keyboard</code>	The stream control block for the keyboard.
<code>rootnode!Rtn_screen</code>	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`.

9.2.1 Stream Input/Output

MCPL 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 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, respectively, when the byte buffer becomes empty on reading or full on output. The procedure `endfn` is called to close the stream.

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 2. If a character

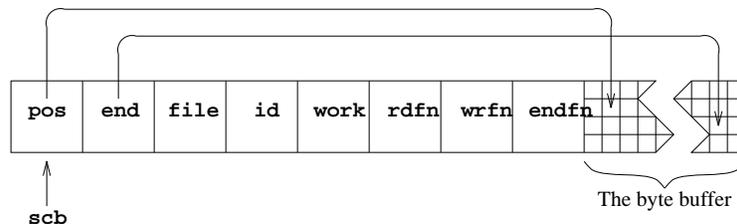


Figure 1: Stream control block structure

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.

```

FUN rdch : =>
  LET pos = cis!Scb_pos
  IF pos < cis!Scb_end DO { cis!Scb_pos := pos+1
                          RETURN cis%pos
                          }
  RETURN (cis!Scb_rdfn)(cis)
.

```

Figure 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 is given in figure 3.

```

FUN unrdch : =>
  LET pos = cis!Scb_pos
  IF pos <= Scb_bufstart RETURN FALSE // Cannot UNRDCH past origin.
  cis!Scb_pos := pos-1
  RETURN TRUE
.

```

Figure 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 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.

```

FUN wrch : ch =>
  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 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.

9.2.2 Input Functions

```

scb := findinput(name)
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 input comes from the keyboard, otherwise `name` is taken to be a filename. If the stream cannot be opened the result is zero.

The call `selectinput(scb)` selects `scb` as the currently selected input stream. It aborts 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 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 succeed the first time after a call of `rdch`. Its main purpose is for use 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.

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.

9.2.3 Output Functions

`scb := findoutput(name)`

This function opens an output stream. If `name` is the string `"*"` then output is to the screen, otherwise `name` is taken to be a file name. If the stream cannot be opened, the result is zero.

`selectoutput(scb)`

This routine selects `scb` as the currently selected output stream. It aborts 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.

`scb := output()`

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

`endwrite()`

This routine closes the currently selected output stream.

`newline()`

`newpage()`

These two routines simply output the newline character (`'\n'`) or the new-page (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)`

These routines output, respectively, the least significant `d` octal or hexadecimal 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 `%5d` is encountered when a value (usually the next argument) is output in the specified format. The substitution items are given in table 5.

Item	Substitution
<code>%s</code>	Write the next argument as a string using <code>writes</code> .
<code>%nt</code>	Write the next argument as a left justified string in a field width of n characters using <code>writet</code> .
<code>%c</code>	Write the next argument as a character using <code>wrch</code> .
<code>%no</code>	Write the next argument as an octal number in a field width of n characters using <code>writeoct</code> .
<code>%nx</code>	Write the next argument as a hexadecimal number in a field width of n characters using <code>writehex</code> .
<code>%nd</code>	Write the next argument as a decimal number in a field width of n characters using <code>writed</code> .
<code>%d</code>	Write the next argument as a decimal number in its natural field width using <code>writen</code> .
<code>%nu</code>	Write the next argument as an unsigned decimal number in a field width of n characters using <code>writeu</code> .
<code>\$\$</code>	Skip over the next argument.
<code>%%</code>	Write the character <code>%</code> .

Figure 5: `writef` substitution items

When a field width (denoted by n) is required, it is specified by a possibly empty sequence of decimal digits. The implementation of `writef` is a good example of how a variadic function can be defined. It can be found in `MCPL/mintcode/sys/MLIB.m`.

9.2.4 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 character slash (/) or backslash (\) is regarded as a file separator and is converted into the appropriate separator character 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 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.

As described above, files may be opened and closed for reading and writing. They may also be deleted or renamed using the the following routines.

```
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.

9.2.5 Command Arguments

This implementation of MCPL incorporates the command language interpreter which is described in Section 10, and, to assist command argument decoding, the functions `rditem`, `rdargs`, `findarg` and `str2numb` are provided.

```
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` and its upper bound is `upb` and returns an integer describing the kind of item read. Table 6 shows the kinds of item read and corresponding results.

Newline characters can occur within a quoted string 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

Example items	Kind of item	Result
"from" "\ntwo words\n"	Quoted string	2
abc 123-45*6	Unquoted string	1
<i>carriage return</i> ; <i>end-of-stream</i>	Terminator	0
	An error	-1

Figure 6: `rditem` results

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 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.

```
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`,

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

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`. It uses the global variable `randseed` as its seed.

`obj := mkobj(upb, fns, x)`

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

```

FUN mkobj :upb, fns, x => // object making function
  LET a = getvec(upb)
  UNLESS a=0 DO
  { !a := fns
    Initobj#(a, x) // Send the InitObj message to the object
  }
  RETURN a

```

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`. It returns the a pointer to the initialised fields vector. If it fails, it returns zero.

9.2.6 Program Loading and Control

In this implementation, the MCPL compiler generates a file of hexadecimal numbers for the compiled code. For instance the compiled form of the `logout` com-

mand:

```
MODULE logout
GET "mcpl.h"
FUN start : => abort 0
.
```

is

```
000007D0 0000000D
0000000D 0000FDDF 6F676F6C 00207475 0000DFDF
72617473 00202074 BA1F2310 7BFF1003 00000000
00000001 0000001C 0000001F
```

The first two words indicate the presence of a “hunk” of code of size 13 words which then follow. The two words following the pattern 0000FDDF contain the `MODULE` name “logout” and the two words following 0000DFDF contain the procedure name “start”. The body of `start` (BA1F2310 7BFF1003) then follows. The remaining 4 words are 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). The following functions are related to the loading and control of programs.

`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 10), 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 `MLIB` and simply calls `start`. Its purpose is to assist debugging by providing a place to set a breakpoint in the command language interpreter (MCLI) just before a command is entered. It 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.

`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 Mintcode system altogether, if `code` is `-1` execution resumes using the faster version of the interpreter (`mintasm`). 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.

`seg1 := 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(seg1)`

This function initialises the global variables defined in the list of program modules given by its argument `seg1`. It returns zero if the global vector was too small, otherwise it returns `seg1`.

`unloadseg(seg1)`

This routine unloads the list of loaded program modules given by `seg1`.

`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.

9.2.7 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 match.

```
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.

9.2.8 Coroutines

MCPL 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 quantities. Occasionally, applications arise where it is 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 so it is natural to represent a coroutine by a pointer to its stack. At the base of each stack there are seven words of system information as shown in figure 8.

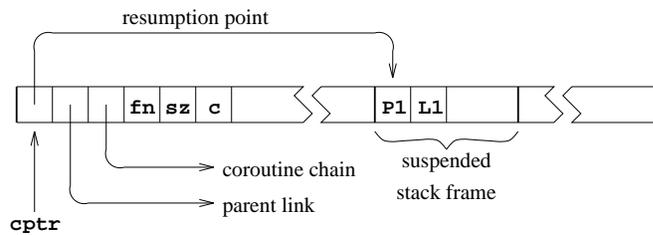
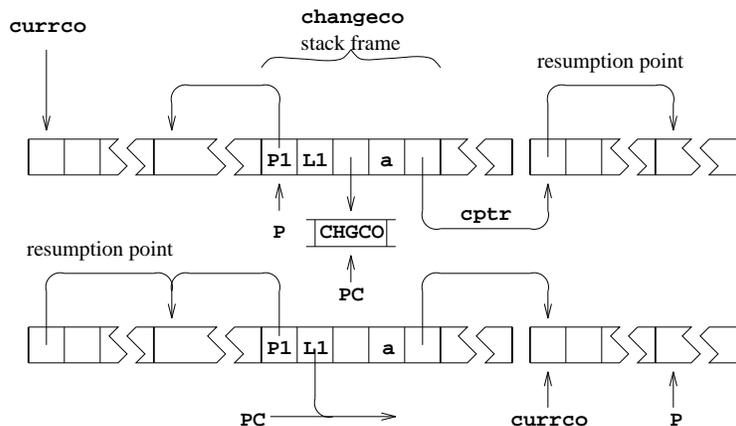


Figure 8: A coroutine stack

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 (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.

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 and H pointer in the current coroutine, and setting new values for the program counter (PC), the P pointer, H pointer and `currco`. This is done by `change(a, cptr)` as shown in figure 9. The function `change` is defined by

Figure 9: The effect of `changepco(a, cptr)`

hand in `MSYSLIB` and its body consists of the single instruction `CHGCO` and as can be seen its effect is somewhat subtle. The only uses of `changepco` 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 transferred 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+7` 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 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.

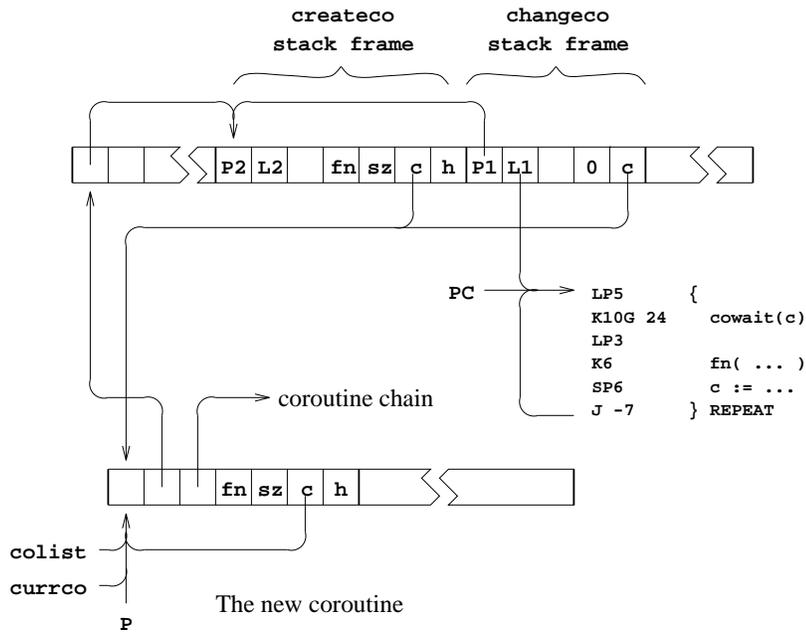


Figure 10: The state just after `changeco(0,c)` in `createco`

```
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.

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

This function is defined in MLIB to provide a convenient method of creating and initialising coroutines. Its definition is as follows:

```

FUN initco : fn, size, a, b, c, d, e, f, g, h, i, j, k =>
  LET cptr = createco(fn, size)
  UNLESS cptr=0 DO callco(cptr, @a)
  RETURN 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.

9.2.9 Hamming's Problem

A following problem permits a neat demonstration of 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 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.

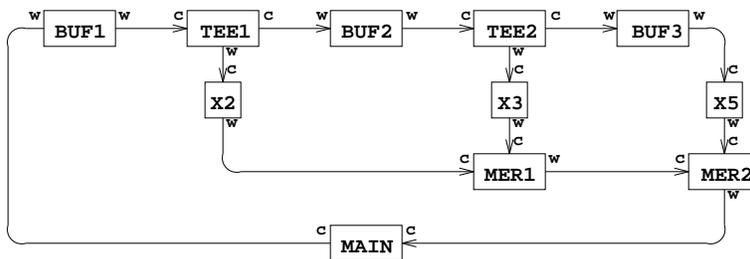


Figure 11: Coroutine data flow

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, respectively. `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 MCPL code for this solution is as follows:

```
GET "mcpl.h"

FUN buf : =>                                     // Body of buf1, buf2 and buf3
  LET val = cwait()
  LET p=0, q=0
  LET v = VEC 200

  { // val = 0 means extract value from the buffer
    // val~= 0 means push value into the buffer
    TEST val THEN { IF p=q+201 DO writef "Buffer full\n"
                    v!(p++ MOD 201) := val
                  }
    ELSE { IF p=q DO writef "Buffer empty\n"
           val := v!(q++ MOD 201)
         }
    val := cwait val
  } REPEAT

FUN tee : [a1, a2] =>                             // Body of tee1 and tee2
  LET in=a1, out=a2
  cwait()                                         // End of initialisation.

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

FUN mul : [a1, a2] =>                             // Body of x2, x3 and x5
  LET k=a1, in=a2
  cwait()                                         // End of initialisation.

  cwait(k * callco(in, 0)) REPEAT

FUN merge : [a1, a2] =>                          // Body of mer1 and mer2
  LET inx=a1, iny=a2
  cwait()                                         // End of initialisation

  LET x=0, y=0, min=0

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

```

FUN start : =>
  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(" %6d", val)
                        UNLESS i MOD 10 DO newline()
                        callco(buf1, val) // Push val into buf1
                        val := callco mer2
                      }

  deleteco buf1; deleteco buf2; deleteco buf3
  deleteco tee1; deleteco tee2
  deleteco x2; deleteco x3; deleteco x5
  deleteco mer1; deleteco mer2
.

```

9.2.10 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.

10 The Command Language

The Command Language Interpreter (MCLI) is a simple interactive interface between the user and the system. Its source code is in `/mintcode/sys/MCLI.m`. It loads and executes previously compiled programs that are normally held in the `mintcode` directory. The source code of most commands can be found in `mintcode/com`. Many of the system provided commands are described in section 10.2. However, any compiled program can be regarded as a command and so the command language can be extended easily by the user.

10.1 Bootstrapping

When the Mintcode System is started, control is passed to the interpreter which, after a few initial checks, allocates vectors for the memory of the mintcode abstract machine machine and the tally vector available for statistics gathering. The mintcode 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 12 is completed by loading the object modules `MSYSLIB`, `MLIB` and `MBOOT`, and initialising the root node, the stack and global vector. Interpretation of mintcode instructions now begins with the mintcode register `PC`, `P` and `G` set as shown in the figure, and `Count` set to `-1`. The other registers are cleared. The first mintcode instruction to be executed is the first instruction of the body of the routine `start` in `MBOOT`, whose source is in `mintcode/sys/MBOOT.m`. 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 stack and global vector shown in figure 12 are used by `start` and form the running environment of the debugger. The MCLI, on the other hand, is provided with a new stack and a separate global vector, thus allowing the debugger to use globals freely without danger of interference from the command language interpreter or from running commands. The global vector of 1000 words is allocated for MCLI and this is shared by the MCLI 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.

Control is passed to the MCLI by means of the call `sys(1,regs)` which recursively enters the interpreter from an initial mintcode 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 MCLI, respectively, `PC` is the location of the first instruction of

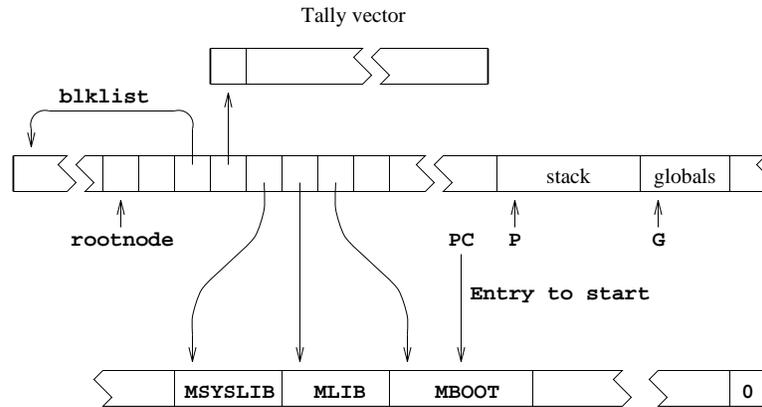


Figure 12: The initial state

`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 been placed in the global vector. Thus, for instance, instead of calling `globin` to initialise the globals defined in `MSYSLIB` and `MLIB`, the following code is used:

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

If a fault occurs during the execution of `MCLI` 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 mintcode registers. A result of zero, signifying successful completion, causes execution of the mintcode system to terminate; however, if a non zero result is returned, the debugger is entered by means of the call `debug(res)`. Note that the mintcode 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 11.

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 initialised and the standard

input and output streams opened before loading the MCLI program by means of the following statement:

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

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

10.2 Commands

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

abort NUMBER

The command: `abort n` causes the MLIB function `abort` to be called with argument `n`. If `n = 0`, it will cause a successful return from the mintcode 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 11.

echo TEXT,N/S

This command will output its first argument `TEXT`. Unless the `N` switch is given, the text will be followed by a newline.

interpreter FAST/S,SLOW/S

This command allows the user to select the fast (`mintasm`) or the slow (`minterp`) version of the interpreter. If no arguments are given the fast one is selected.

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

This command outputs the state of the mintcode 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.

preload ,,,,,,,

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

prompt PROMPT

This command allows the user to change the prompt string. The default prompt format is: `"%d> "`, which will output the time in milliseconds used by the previous command.

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 mintcode 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 (mintasm) 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 switch argument **N** causes line numbers to be added.

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

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

11 The Debugger

When the mintcode system starts up, control first passes to **MBOOT** which initialises the system and creates a running environment for the command language interpreter (**MCLI**). This is run by a recursive invocation of the interpreter and so when faults occur control returns to **MBOOT** 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 0Bn 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
* =                Display the current value
* <               Shift left one place
* =                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 mintcode 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=          32924          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          42844          42844          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, \$0, \$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 mintcode instructions using the commands I and N. For example:

```

* g4=      clihook          Get the entry to clihook
* i  21352:  LP3          Load the local at P3
* n  21353:  K4G  1          Call global 1, incrementing P by 4
* n  21355:  J  21359          Jump to 21359
* n  21357:  L0          Load 0
* n  21358:  RAISE          Raise exception 0 PATERR
* n  21359:  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
10>

```

The X command could have been used since it is a shorthand for G4B9C. The function `clihook` is defined in `MLIB` 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  21352:  LP3
*

```

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

```
* \A=          0 B=          0   21353:   K4G  1
* \A=          0 B=          0   42768:   LLP  4
* \A=       42904 B=          0   42770:   SP3
* \A=       42904 B=          0   42771:    SP  89
* \A=       42904 B=          0   42773:    L  80
* \A=          80 B=       42904   42775:    SP  90
* \A=          80 B=       42904   42777:   LLL 42720
* \A=       42720 B=          80   42779:    LG  78
* \A=      rdargs B=       42720   42781:    K  85
* \A=       42720 B=       42720   23660:   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:      40655
   A=       42783 B=          1   42783:   CTA
*
```

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:      42783
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 mintcode system.

12 Installation

A free distribution of MCPL is available via my World Wide Web Home Page (<http://www.cl.cam.ac.uk/users/mr/>) 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 distribution is still under construction and is in an unpolished state, however, it works well enough to run several demonstration programs. It is ultimately meant to provide a machine independent interpretive version of MCPL. The interpreter is implemented in C, but its efficiency will be improved when an

assembly code version of part of the system is completed, as has been done for the BCPL Cintcode system.

This distribution is being developed under Linux, but should be (easily?) transferable to DEC Alpha machines, Mips R2000/3000 machines, Sun4s, Sun/SPARCs and 386/486 machines under MSDOS/Windows, MAC or OS/2. The simplest installation is for Linux machines.

12.1 Linux Installation

This section describes how to install the MCPL Mintcode on an IBM PC running Linux.

- First install the BCPL Cintcode system (`bcpl.targz`) including the latest version of BCPL/bcplprogs (`bcplprogs.tar.gz`), available from my WWW Homepage.
- The following directory structure is required:

```

|
|--BCPL                (from bcpl.targz or bcpl.zip)
|  |
|  |--cintcode         source and compiled code + headers
|  |
|  |--bcplprogs       various BCPL demonstration programs
|  |
|  |--natbcpl         experimental native code version
|
|--MCPL                (from mcpl.tgz or mcpl.zip)
|  |
|  |--mcplcomp        C, BCPL and MCPL source code
|  |
|  |--mintcode        compiled code + header files
|  |
|  |--mcplprogs       various MCPL demonstration programs
|  |
|  |--natmcpl        experimental native code version

```

First install the BCPL system (`bcpl.tgz` or `bcpl.zip`) which is available from my WWW Homepage. Make sure you complete the installation of BCPL including the setting of `PATH` and `BCPLPATH` (see the BCPL distribution README file), before attempting to install MCPL.

Next create a shell (xterm) window and create the MCPL directory in the right place and enter it.

```

mkdir MCPL
cd MCPL

```

- Copy the files `mcpl.tgz` into this directory and extract the files.

```
cp ...../mcpl.tgz .
gunzip -c mcpl.tgz | tar xvf -
```

This will create the directories `mcplcomp`, `mintcode`, `mcplprogs` and `natmcpl`, and fill them with useful material. The directory `mcplprogs` contains collections of demonstration programs.

Alternatively, use `mcpl.zip` and appropriate decompression step.

- Change directory to `mcplcomp`.

```
cd mcplcomp
```

This will compile various files putting the results (`minterp`, `MBOOT`, `MLIB` and `MCLI`) in `mintcode`.

- Add this directory to the shell variables `PATH` and `MCPLPATH`. This could possibly be done by adding the following to your `.profile` file:

```
export PATH=$PATH:$(HOME)/MCPL/mintcode
export MCPLPATH=$(HOME)/MCPL/mintcode
```

- To compile an MCPL program (eg `MCPL/mcplprogs/demos/queens.m`) change to directory `MCPL/mcplprogs/demos`, enter the BCPL system and invoke the MCPL compiler by typing:

```
cd ....MCPL/mcplprogs/demos -- get into demos directory
cd ....MCPL/mcplprogs/demos -- get into demos directory
cinterp                    -- enter the BCPL system
c mp queens                 -- compile the queens program
```

This will compile `queens.m` leaving the compiled code `queens` in the current working directory.

- To execute the queens program, first create a new shell (xterm) window then type:

```
cd ../MCPL/mcplprogs/demos -- get into demos directory
minterp                    -- enter the MCPL system
queens                     -- execute the queens program
```

Some other example programs worth trying in demos are:

```
primes    -- print primes less than 1000
coins     -- how many ways to represent a given sum of money
fridays   -- shows that the 13th is most often a friday
turing    -- a Turing machine demo
crdemo    -- a coroutine demo
eval      -- a lambda evaluator
fft       -- a demo fft program using modulo arithmetic
pento     -- finds 2339 solutions of the pentominoes problem
           -- allow 20 minutes or so!
```

There are many other demonstration programs in MCPL/mcplprogs. To leave the MCPL mintcode system type `logout` or `(^C)`.

- The MCPL programs that are part of the system are: MBOOT, MLIB and MCLI. These reside in MCPL/mcplcomp/sys and are compiled by the following commands when in MCPL/mcplcomp.

```
cinterp
c ms MBOOT
c ms MLIB
c ms MCLI
```

System commands are in MCPL/mintcode/com may be compiled using `mc`.

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

The MCPL compiler is a BCPL program in MCPL/mcplcomp. Its called `mcpl.b`, and it GETs `mcplsyn.b` and `mcpltrn.b` from the same directory. It is compiled by the `make` command mentioned above.

Read the documentation in `mintcode/doc` and any `README` files you can find. A postscript version of the current version of this MCPL manual is available from my home page.

12.2 Installation on Other Machines

This has not yet been tested but will be similar to installing BCPL on other machines.

12.3 The Native Code Version

The directory `MCPL/natmcpl` contains files that make up the native code versions of MCPL for Linux. Look at the file `README` in `MCPL/natmcpl/doc` to see how to install and use the system. `README` in `MCPL/natmcpl/doc` to see how to install and use the system.

`MCPL/natmcpl` contains a version of a MCPL system compiled into native assembly language, currently for Intel 386/486/Pentium based machines running under Linux. It uses a very simple codegenerator (`cvmial386`) that translates the MCPL internal assembly language (MIAL) into Gnu assembler for the 386. MCPL is translated to MIAL by a BCPL program called `mcpl2mial`, whose source code is in `BCPL/bclpprogs/mcpl2mial.b`. MIAL is sufficiently close to machine code to make it possible to modify `cvmial386` for other architectures in about three days.

13 Example Programs

13.1 Coins

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

```
GET "mcpl.h"

FUN ways
: 0,      ?  => 1
: ?,      [0] => 0
: s, coins[>s] => ways(s, @ coins!1)
: s, coins[v ] => ways(s, @ coins!1) + ways(s-v, coins)
.

FUN t
: sum => writef("Sum = %3d,   Ways = %4d\n",
               sum,       ways(sum, [50, 20, 10, 5, 2, 1, 0]))
.

FUN start : => t 0; t 1; t 5; t 11; t 20; t 50
RETURN 0
.
```

13.2 Primes

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

```
GET "mcpl.h"
MANIFEST Upb = 1000
FUN start : =>
    writef "\nTable of prime numbers\n\n"
    LET count=0, isprime=VEC Upb
    FOR i = 2 TO Upb DO isprime!i := TRUE
    FOR p = 2 TO Upb IF isprime!p DO
    { LET i = p*p
      UNTIL i>Upb DO { isprime!i := FALSE
                      i += p
                      }
      writef(" %3d", p)
      IF ++count MOD 10 = 0 DO writef "\n"
    }
    writef "\nEnd of output\n"
    RETURN 0
.
```

13.3 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 "mcpl.h"
STATIC count, all
FUN try
: ?, =all, ? => count++
: ld, row, rd => LET poss = ~(ld | row | rd) & all
                 WHILE poss DO
                 { LET bit = poss & -poss
                   poss -= bit
                   try( (ld|bit)<<1, row|bit, (rd|bit)>>1 )
                 }
.
```

```

FUN start : =>
  all := 1
  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("There are %5d solutions to %2d-queens problem\n",
          count,          i )
    all := 2*all + 1
  }
  RETURN 0
.

```

13.4 Fridays

The following program prints a table of how often the 13th day of the month lies on each day of the week over a 400 year period.

```

GET "mcpl.h"

MANIFEST Mon = 0, Sun = 6,
          Jan = 0, Feb = 1, Dec = 11

STATIC
count      = [0,0,0,0,0,0,0],
days      = 0,
daysinmonth = [31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
dayname    = ["Monday", "Tuesday", "Wednesday", "Thursday",
              "Friday", "Saturday", "Sunday"]
.

FUN start : =>
  FOR year = 1973 TO 1973 + 399 DO
  { daysinmonth!Feb := febdays year
    FOR month = Jan TO Dec DO { LET day13 = (days+12) MOD 7
                              (count!day13) ++
                              days += daysinmonth!month
                              }
    }

  FOR day = Mon TO Sun DO writef(" %3d %ss\n", count!day, dayname!day)
  RETURN 0
.

FUN febdays : year => year MOD 400 = 0 -> 29,
                    year MOD 100 = 0 -> 28,
                    year MOD 4   = 0 -> 29,
                    28
.

```

13.5 Prover

This program is a very free translation of the LISP version of the Wang Algorithm given in the LISP 1.5 book. It checks whether formulae in propositional logic are


```

//***** Lexical Analyser *****
STATIC
  strp, ch, nch, token, lexval

FUN lex_init : str => strp := str; rch(); rch().

FUN rch : => ch, nch := nch, %strp
  UNLESS nch=0 DO strp++
.

FUN lex : => MATCH (ch, nch)
: ' ' | '\n' => rch(); lex()
: 'A'..'Z'   => token := Id; lexval := rch()
: '('       => token := Lparen; rch()
: ')'      => token := Rparen; rch()
: '~'      => token := Not; rch()
: '&'      => token := And; rch()
: '|'      => token := Or; rch()
: '='      => token := Eqv; rch()
: '->', '>' => token := Imp; rch(); rch()
: 0        => token := Eof
:          => RAISE E_syntax
.
.

//***** Syntax Analyser *****

// A .. Z --> [Id, 'A'] .. [Id, 'Z']
// ~x      --> [Not, x]
// x & y   --> [And,x,y]
// x | y   --> [Or,x,y]
// x -> y  --> [Imp,x,y]
// x = y   --> [Eqv,x,y]

FUN parse : str => lex_init str
  LET tree = nexp 0
  //writef("Parse tree:\n")
  //prtree (tree, 0, 20)
  //newline()
  tree
.

FUN prim : => MATCH token
: Id     => LET a = mk2(Id, lexval)
  lex()
  a
: Lparen => LET a = nexp 0
  UNLESS token=Rparen RAISE E_syntax
  lex()
  a
: Not    => mk2(Not, nexp 3)
: ?      => RAISE E_syntax
.
.

```

```

FUN nexp : n => lex(); exp n
.

FUN exp : n =>
  LET a = prim()
  MATCH (token, n)
  : And, <3 => a := mk3(And, a, nexp 3)
  : Or, <2 => a := mk3(Or , a, nexp 2)
  : Imp, <1 => a := mk3(Imp, a, nexp 1)
  : Eqv, <1 => a := mk3(Eqv, a, nexp 1)
  :      => RETURN a
  . REPEAT
.

//***** Space Allocation *****

STATIC
  spacev, spacep

FUN mk_init : upb =>
  spacev := getvec upb
  IF spacev=0 RAISE E_space
  spacep := @ spacev!upb
.
FUN mk_close : => freevec spacev.
FUN mk1 : x => !---spacep := x; spacep.
FUN mk2 : x, y => mk1 y; mk1 x.
FUN mk3 : x, y, z => mk1 z; mk1 y; mk1 x.

//***** Main Program *****

FUN try : e =>
  { writef("\nTesting: %s\n", e)
    mk_init 100_000
    TEST pr(0, 0, 0, [parse e, 0])
    THEN writef("----- It is a tautology\n")
    ELSE writef("----- It is NOT a tautology\n")
  } HANDLE : E_syntax => writef "Bad Syntax\n"
           : E_space  => writef "Insufficient space\n"
.
  mk_close()
.

// Propositional examples supplied by Larry Paulson
// and modified by MR

FUN start : =>
  writef "\nAssociative laws of & and |\n"
  try "(P & Q) & R = P & (Q & R)"
  try "(P | Q) | R = P | (Q | R)"

  writef "\nDistributive laws of & and |\n"
  try "(P & Q) | R = (P | R) & (Q | R)"
  try "(P | Q) & R = (P & R) | (Q & R)"

```

```

writef "\nLaws involving implication\n"
try "(P|Q -> R) = (P->R) & (Q->R)"
try "(P & Q -> R) = (P-> (Q->R))"
try "(P -> Q & R) = (P->Q) & (P->R)"

writef "\nClassical theorems\n"
try "P | Q -> P | ~P & Q"
try "(P->Q)&( ~P->R) -> (P&Q | R)"
try "P & Q | ~P & R = (P->Q) & (~P->R)"
try "(P->Q) | (P->R) = (P -> Q | R)"
try "(P = Q) = (Q = P)"

/* Sample problems from F.J. Pelletier,
   Seventy-Five Problems for Testing Automatic Theorem Provers,
   J. Automated Reasoning 2 (1986), 191-216.
*/

writef "\nProblem 5\n"
try "((P|Q)->(P|R)) -> (P|(Q->R))"

writef "\nProblem 9\n"
try "((P|Q) & ( ~P | Q) & (P | ~Q)) -> ~( ~P | ~Q)"

writef "\nProblem 12. Dijkstra's law\n"
try "((P = Q) = R) -> (P = (Q = R))"

writef "\nProblem 17\n"
try "(P & (Q->R) -> S) = (( ~P | Q | S) & ( ~P | ~R | S))"

writef "\nFALSE GOALS\n"
try "(P | Q -> R) = (P -> (Q->R))"
try "(P->Q) = (Q -> ~P)"
try " ~P->Q -> (Q = P)"
try "(P->Q) -> Q -> P"
try "((P | Q) & (~P | Q) & (P | ~Q)) -> ~(~P | Q)"

writef "\nIndicates need for subsumption\n"
try "((P & (Q = R)) = S) = (( ~P | Q | S) & ( ~P | ~R | S))"

// Prove that the circuit
// -----
// X --| NOT |--> Y
// -----
// is equivalent to:
// -----
// A --| NOT |--B--| NOT |--C--| NOT |--> D
// -----
writef "\nProof of the correctness of a circuit\n"
try "(Y=~X) & ((D=~C) & (C=~B) & (B=~A)) & (X=A) -> (Y=D)"
RETURN 0

```

```

/*****
*
*          Print Tree Functions and Data
*
*   prtree(tree, depth, maxdepth)
*   prlinev
*
*****/

STATIC
  prlinev = VEC 50

FUN prtree : x, depth, maxdepth =>

  LET opstr=0, upb=0

  IF x=0 DO { writef "Nil"; RETURN }

  { MATCH x
    : [Id, ch]           => writef("%c", ch);      RETURN
    : [And, x, y]        => opstr, upb := "And",    2
    : [Or, x, y]         => opstr, upb := "Or",     2
    : [Imp, x, y]        => opstr, upb := "Imp",    2
    : [Eqv, x, y]        => opstr, upb := "Eqv",    2
    : [Not, x]           => opstr, upb := "Not",    1
    :                    => opstr, upb := "Unknown", 0
  }

  IF depth=maxdepth DO { writef("Etc"); RETURN }

  writef("Op:%s", opstr)

  FOR i = 1 TO upb DO { newline()
                      FOR j=0 TO depth-1 DO writes( prlinev!j )
                      writes("*-")
                      prlinev!depth := i=upb-> " ", "! "
                      prtree(x!i, depth+1, maxdepth)
                    }
}

```

13.6 Eval

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

```

GET "mcpl.h"

MANIFEST

Eof=0, Id, Num, Times, Div, Pos, Neg, Plus, Minus,
Eq, Cond, Lam, Ap, Y,

Syntax=1, Lookup, Eval

STATIC spacep

FUN lookup
: ?,          0 => RAISE Lookup
: n, [=n, val, ?] => val
: n, [ ?,   ?, e] => lookup(n, e)
.

FUN eval
: [Id, x],      e => lookup(x, e)
: [Num, k],     ? => k
: [Pos, x],     e => eval(x, e)
: [Neg, x],     e => - eval(x, e)
: [Times,x,y], e => eval(x, e) * eval(y, e)
: [Div,x,y],   e => eval(x, e) / eval(y, e)
: [Plus,x,y],  e => eval(x, e) + eval(y, e)
: [Minus,x,y], e => eval(x, e) - eval(y, e)
: [Eq,x,y],    e => eval(x, e) = eval(y, e)
: [Cond,b,x,y], e => eval(b, e) -> eval(x, e), eval(y, e)
: [Lam,x,body], e => mk3(x, body, e)
: [Ap,x,y],    e => MATCH eval(x, e)
                  : [bv, body, env] =>
                    eval(body, mk3(bv, eval(y, e), env))
.

: [Y, exp],    e =>
  LET bigf = eval(exp, e)
  MATCH bigf // bigf should be a closure whose body is an
              // abstraction eg Lf Ln n=0 -> 1, n*f(n-1)
  : [f, [Lam, n, body], env] =>
    LET res = mk3(n, body, 0) // environment to be filled in later
    LET env1 = mk3(f, res, env) // make environment including f
    res!2 := env1 // fill in environment component of closure
    RETURN res // return this closure
.

: ?,          ? => RAISE Eval
.

```

```

// 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]

STATIC strp, ch, nch, token, lexval

FUN rch
: => ch := nch
      nch := %strp
      UNLESS nch=0 DO strp++
.
FUN parse
: s => strp := s
      rch(); rch()
      nexp 0
.
FUN lex
: => MATCH (ch, nch)

      : '->', '>' => token := '->'; rch(); rch()

      : ' ' | '\n', ? => rch(); lex()

      : '(' | ')' | '*' | '/' | '+' | '-' | 'L' | 'Y' | '=' | ',',
        ? => token := ch; rch()

      : 'a'..'z', ? => token, lexval := Id, ch; rch()

      : '0'..'9', ? => token, lexval := Num, ch-'0'
        { rch()
          MATCH ch
            : '0'..'9' => lexval := 10*lexval+ch-'0'
            : ?       => RETURN
          } REPEAT

      : 0, ? => token := Eof

      : ?, ? => RAISE Syntax
.

```

```

FUN prim
: => MATCH token
  : Id   => LET a = mk2(Id, lexval)
          lex()
          a
  : Num  => LET a = mk2(Num, lexval)
          lex()
          a
  : 'Y'  => mk2(Y, nexp 6)
  : 'L'  => lex()
          UNLESS token=Id RAISE Syntax
          LET bv = lexval
          mk3(Lam, bv, nexp 0)
  : '('  => LET a = nexp 0
          UNLESS token=')' RAISE Syntax
          lex()
          a
  : '+'  => mk2(Pos, nexp 3)
  : '-'  => mk2(Neg, nexp 3)
  : ?    => RAISE Syntax
.

FUN nexp : n => lex(); exp n

FUN exp
: n => LET a = prim()
      MATCH (token, n)
      : '(' | Num | Id,
        <6 => a := mk3( Ap, a, exp 6)
      : '*', <5 => a := mk3(Times, a, nexp 5)
      : '/', <5 => a := mk3( Div, a, nexp 5)
      : '+', <4 => a := mk3( Plus, a, nexp 4)
      : '-', <4 => a := mk3( Minus, a, nexp 4)
      : '=', <3 => a := mk3( Eq, a, nexp 3)
      : '->', <1 => LET b = nexp 0
                    UNLESS token=', ' RAISE Syntax
                    a := mk4(Cond, a, b, nexp 0)
      : '?', ? => RETURN a
      . REPEAT
.

FUN mk1 : a          => !---spacep := a; spacep.
FUN mk2 : a, b       => mk1(b); mk1(a).
FUN mk3 : a, b, c    => mk1(c); mk1(b); mk1(a).
FUN mk4 : a, b, c, d => mk1(d); mk1(c); mk1(b); mk1(a).
FUN wrs : s => writef("%s\n", s).
FUN wrn : n => writef("%d\n", n).
FUN try : e => { newline()
                wrs e
                spacep := @ (VEC 1000)!1000
                wrn ( eval(parse e, 0) )
              } HANDLE : Syntax => wrs "Bad syntax"
                    : Lookup => wrs "Bad lookup"
                    : Eval   => wrs "Bad eval"
.

```

```

FUN start : =>
  try "(Lx x) (Ly y) 99"
  try "1234"
  try "x"
  try "100+23"
  try "1=2 -> 3, 4"
  try "(Lx x+1) 2"
  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))) 10"
RETURN 0

```

13.7 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 "mcpl.h"

MANIFEST
  Modulus = #x10001, // 2**16 + 1

//Omega = #x00003, N = #x10000,
//Omega = #x0ADF3, N = #x01000,

Omega = #x096ED, N = #x00400,

//Omega = #x08000, N = #x00010,
//Omega = #x0FFF1, N = #x00008,

// Omega and N are chosen so that: Omega**N = 1

  Upb      = N - 1, // N is a power of 2
  MSB      = N>>1,
  LSB      = 1

STATIC v, w

FUN start : =>
  LET a = 1 // First check Omega has the right properties
  FOR i = 1 TO Upb DO
  { a := mul(a, Omega)
  IF a=1 DO writef("Omega***%d = 1\n", i)
  }
  UNLESS mul(a, Omega)=1 DO writef("Omega***%d ~= 1\n", N)

  v := VEC Upb
  w := VEC Upb

  FOR i = 0 TO Upb DO v!i := i

```

```

    pr(v, 15)
// prints -- Original data
//   0   1   2   3   4   5   6   7
//   8   9  10  11  12  13  14  15

    dofft v

    pr(v, 15)
// prints -- Transformed data
// 65017 26645 38448 37467 30114 19936 15550 42679
// 39624 42461 43051 65322 18552 37123 60445 26804

    invfft v

    pr(v, 15)
// prints -- Inverse transform of transformed data
//   0   1   2   3   4   5   6   7
//   8   9  10  11  12  13  14  15
    RETURN 0
.

FUN dofft : v =>
    w!0 := 1 // Nth roots of unity
    FOR i = 1 TO Upb DO w!i := mul(w!(i-1), Omega)
    fft(N, v, 0, MSB)
    reorder(v, v, MSB, LSB)
.

FUN invfft : v =>
    w!0 := 1 // inverse Nth roots of unity
    FOR i = 1 TO Upb DO w!i := ovr(w!(i-1), Omega)
    fft(N, v, 0, MSB)
    reorder(v, v, MSB, LSB)
    FOR i = 0 TO Upb DO v!i := ovr(v!i, N)
.

FUN reorder
: p, <p, 0, ? => RETURN
: [x], [y], 0, ? => x, y := y, x
: p, q, a, b => LET a1=a>>1, b1=b<<1
                reorder(@p!a, @q!b, a1, b1)
                reorder(p, q, a1, b1)
.

FUN fft
: nn, v, pp, msb => LET n=nn>>1, p=pp>>1
                    FOR i = 0 TO n-1 DO butterfly(@v!i, @v!(i+n), w!p)
                    IF n=1 RETURN
                    fft(n, @v!0, p, msb)
                    fft(n, @v!n, msb+p, msb)
.

FUN butterfly
: [x], [y], wk => LET t = mul(y, wk)
                  x, y := add(x, t), sub(x, t)
.

```

```

FUN pr : v, upb =>
  IF upb > Upb DO upb := Upb
  FOR i = 0 TO upb DO
    { writef("%5d ", v!i)
      IF i MOD 8 = 7 DO writef "\n"
    }
  writef "\n"
.

FUN dv
: 1, m,      ?, ? => m
: 0, m,      ?, n => m-n
: a, m, (>a)b, n => dv(      a,          m, b MOD a, m*(b/a)+n)
: a, m,      b, n => dv(a MOD b, n*(a/b)+m,          b,          n)
.

FUN inv
: x => dv(x, 1, Modulus-x, 1)
.

FUN ovr
: x, y => mul(x, inv(y))
.

FUN mul
: 0, ? => 0
: x, y => (x&1)=0 -> mul(x>>1, add(y,y)),
          add(y, mul(x>>1, add(y,y)))
.

FUN add
: x, y => LET a = x+y
          0<=a<Modulus -> a, a-Modulus
.

FUN sub
: x, y => add(x, Modulus-y)
.

```

13.8 Turing

The following program simulates the execution of a Turing Machine running a Turing program that is designed to multiply two numbers together in unary. The call: `turing("A11", 'B', "111A")` will run the machine with the reading head positioned over the character B and with a given initial setting of the left and right portions of the tape. After several steps it will print:

A 1 1 1 1 1 1 [A]

indicating the the head is positioned on the right hand A, with the answer repre-

sented by the six 1s.

```

GET "mcpl.h"

FUN start : =>
  writef "Turing entered\n\n"

  { mk_init 200000
    turing(s0, "A11", 'B', "111A")
  } HANDLE : E_mk, mess => writef("Error: %s\n", mess).

  mk_close()
.

// Space Allocation *****
STATIC spacev=0, spacep=0

MANIFEST E_mk

FUN mk_init : upb =>
  spacev := getvec upb
  IF spacev=0 RAISE (E_mk, "Unable to allocate workspace")
  spacep := @ spacev!upb
.
FUN mk_close : => freevec spacev
.
FUN mk2 : x, y => LET p = @ spacep!-2
                  IF p<spacev RAISE (E_mk, "Out of space")
                  p!0, p!1, spacep := x, y, p
                  RETURN p
.

// Turing Machine Implementation *****
STATIC ltape, ch, rtape

FUN turing : init_state, lstr, char, rstr =>
  ltape, ch, rtape := 0, char, 0

  LET i = 0
  WHILE rstr%i DO i++
  WHILE i DO rtape := mk2(rtape, rstr%--i)
  WHILE lstr%i DO ltape := mk2(ltape, lstr%i++)

  pr()
  init_state ch
.

```

```

FUN right : c =>
  IF rtape=0 DO rtape := mk2(0, ' ')

  MATCH rtape
  : [link, k] => link, ltape, rtape, ch, k :=
                ltape, rtape, link, k, c
                pr()
                RETURN ch
.
.
FUN left : c =>
  IF ltape=0 DO ltape := mk2(0, ' ')

  MATCH ltape
  : [link, k] => link, rtape, ltape, ch, k :=
                rtape, ltape, link, k, c
                pr()
                RETURN ch
.
.
FUN halt : c => ch := c; pr()
.

// Output Routines *****

FUN pr : =>
  prb ltape           // Print the tape on the left
  writef("[%c]", ch) // Print the current character
  prf rtape           // Print the tape on the right
  newline()
.
FUN prb               // Print list of chars backwards
: 0 => RETURN
: [chs, ch] => prb chs; wrch ' '; wrch ch
.
FUN prf               // Print list of chars forwards
: 0 => RETURN
: [chs, ch] => wrch ch; wrch ' '; prf chs
.

```

```
// Turing Machine Definition *****
FUN s0 : '1' => s1 (right '0')
        : 'A' => s2 (right ' ')
        : c => s0 (left c )
.
FUN s1 : 'A' => s3 (left 'A')
        : 'X' => s1 (right '1')
        : c => s1 (right c )
.
FUN s2 : 'A' => s5 (right 'A')
        : ? => s2 (right ' ')
.
FUN s3 : 'B' => s0 (left 'B')
        : '1' => s4 (right 'X')
        : c => s3 (left c )
.
FUN s4 : ' ' => s3 (left 'X')
        : c => s4 (right c )
.
FUN s5 : ' ' => halt 'A'
        : 'X' => s5 (right '1')
        : c => s5 (right c )
.
```

/*

The above Turing Machine will multiply two numbers given in unary. It is exercised by the call:

```
turing(s0, "A11", 'B', "111A")
```

which will trace the execution of the machine as follows:

```
A 1 1[B]1 1 1 A
A 1[1]B 1 1 1 A
A 1 0[B]1 1 1 A
A 1 0 B[1]1 1 A
A 1 0 B 1[1]1 A
A 1 0 B 1 1[1]A
A 1 0 B 1 1 1[A]
A 1 0 B 1 1[1]A
A 1 0 B 1 1 X[A]
A 1 0 B 1 1 X A[ ]
A 1 0 B 1 1 X[A]X
A 1 0 B 1 1[X]A X
```

... many lines until ...

```

A O O[B]X X X A X X X X X X
A O [O]B X X X A X X X X X X
A [O]O B X X X A X X X X X X
[A]O O B X X X A X X X X X X
  [O]O B X X X A X X X X X X
    [O]B X X X A X X X X X X
      [B]X X X A X X X X X X
        [X]X X A X X X X X X
          [X]X A X X X X X X
            [X]A X X X X X X
              [A]X X X X X X
                A[X]X X X X X
                  A 1[X]X X X X
                    A 1 1[X]X X X
                      A 1 1 1[X]X X
                        A 1 1 1 1[X]X
                          A 1 1 1 1 1[X]
                            A 1 1 1 1 1 1[ ]
                              A 1 1 1 1 1 1[A]

```

i.e. 2 times 3 equals 6
*/

References

- [CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, Berlin, 1981.
- [HS87] S.P. Harbison and G.L. Steele. *C – A Reference Manual*. Prentice-Hall, 1987.
- [Pau91] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.
- [RWS80] M. Richards and C. Whitby-Stevens. *BCPL - the language and its compiler*. Cambridge University Press, Cambridge, 1980.