Technical Report

UCAM-CL-TR-256
ISSN 1476-2986

Number 256

MCPL programming manual

Martin Richards

May 1992

ISSN 1476-2986

MCPL Programming Manual

by

Martin Richards
mr@uk.ac.cam.cl

Computer Laboratory
University of Cambridge

18 May 1992

Abstract

MCPL is a systems programming language having much in common with BCPL but augmented by the pattern matching ideas of both ML and Prolog. Unlike ML, MCPL is typeless, runs using 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. For efficiency, MCPL uses its own function calling sequence, however a convenient mechanism for mixing MCPL and C programs is provided.

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

This document gives a complete definition of the language and includes, at the end, several examples programs to demonstrate its capabilities.

1 Introduction

The concepts underlying MCPL originates from my experience of using BCPL[1] for the past 25 years and from lessons learnt from using and teaching languages such as ML[2], Prolog[3] and C[4]. Indeed the name MCPL incorporates the letter 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 the typlessness nature of the language causing all expressions to yield values 32 bits long. Even though normal vectors elements have this size, an operator is provided to allow convenient access to strings of packed characters.

As an introductory example, consider the following program:

```
GET "mcpl.h"

FUN start : =>
    printf "Enter three lengths: "
    LET a, b, c = readn(), readn(), read()
    printf("\nLengths entered: %d %d %d", a, b, c)
    printf("\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,    c    => c>a+b        -> "not a",
                   a=c          -> "an equilateral",
                   a=b | b=c    -> "an isosceles",
                   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, which is followed by the function name and a list of pattern match items that successively test the function arguments. The function start has an empty pattern indicating that it is a parameterless function, and, as can be seen, it will output a message, read three numbers and then output two messages about the triangle entered.

The function sort_of_triangle returns a string that depends on its three arguments. When the second argument is less than the value of 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. The compiler notices that both calls are tail recursive, and optimises 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, in the majority of applications, data can be represented satisfactorily by means of integers, strings, bit patterns and pointers to vectors composed of these kinds of objects. 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 many situations where the absence of types is a great advantage – it does, for instance, greatly simplify the pattern matching mechanism that forms so significant part of the language.

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 (or activation record) 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. Input and output is 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 the next occurrence of */. A comment is treated as if it were a single space character. Comments may not occur in the middle of multi-character symbols such as identifiers or string constants.

3 Expressions

Expressions are composed of names, constants and expression operators and may be grouped, if necessary, using parentheses. The precedence and associativity of the different expression constructs is given in Section 3.9.

3.1 Names and Constants

Syntactically a name is of a sequence of letters, digits and underlines starting with a letter. If the name starts with a capital letter, it denotes a constant that must have been declared within a previous `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 entry point, depending on how it was declared. The value of a name is always a 32 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 to four characters, followed by a second single quote. The characters are packed into 8 bit bytes to form a 32 bit value, padded on the left, if necessary, with zeroes. 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:

\n	A single character interpreted by the system as end-of-line.
\p	A newpage character.
\s	A space character.
\b	A backspace character.
\t	A tab character.
\^*c*	The control character *c*, for any appropriate *c*.
\ddd	The single character with number *ddd* (one or more decimal digits denoting an integer in the interval [0,255]).
\"	"
\'	'
\\\\	\\
\f..f\\	This sequence is ignored, where *f..f* stands for a sequence of one or more formatting characters.

The formatting characters are space, tab, newline and newpage.

A string constant consists of a sequence of zero or more characters enclosed within quotes ("). Both string and character constants use the character escape mechanism described above. The value of a string is a pointer to the place in memory where the characters are packed. A zero byte is appended to mark the end of the string. This makes MCPL and C strings compatible.

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, a, b, ptr = token, ?, ?, [3,?,?,?]
```

3.2 Vectors and Tables

An expression of the form:

```
[ E0 ,..., En ]
```

returns a pointer to n+1 consecutive locations of dynamic memory initialised with the values of the expressions E0,..,En. 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 below. The initialising expressions E0,..,En 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 3.10) 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 [ SK0 ,..., SKn ]
```

where SK0,..,SKn are static constant expressions (see Section 3.11). 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. Some example calls are as follows:

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

An empty argument list must to be specified explicitly using empty parentheses `()`. Multiple arguments must be enclosed in parentheses, but a single argument does not need them, provided it is a name, a number, a string, `TRUE`, `FALSE` or an initialised dynamic vector. The last example illustrates a call in which the function is specified by an expression.

If a function is called using a name that was declared to be a C function by an `EXTERNAL` declaration, then the C calling sequence is used passing the arguments as specified by the `EXTERNAL` declaration (see Section 6.1). 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 is accessible from other modules, and if it was specified by an `EXTERNAL` declaration to be a C function then it will be compatible with the C calling sequence. This scheme allows for convenient mixing of C and MCPL programs.

If a function call occurs in the context of an expression then it is assumed to return a 32 bit result.

3.4 Postfixed Expression Operators

Expressions of the form: `E++`, `E+++`, `E--` or `E---` cause the location specified by `E` to be incremented or decremented by one or four. The result of the expression is its original value, before modification. The operator `++` increments by one, `+++` increments by four, `--` decrements by one and `---` decrements by four. 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, and pointers to adjacent words differ by four.

3.5 Prefixed Expression Operators

Expressions of the form: `++E`, `+++E`, `--E` or `---E` cause the location specified by `E` to be incremented or decremented by one or four. The result of the expression is its value after modification. As with the post fixed operators, `++` increments by one, `+++` increments by four, `--` decrements by one and `---` decrements by four, and they work as expected with pointers.

An expression of the form !E returns the contents of the 32 bit memory location pointed to by the value of E, and an expression of the form %E returns an unsigned integer equal to the 8 bit byte pointed to by the value of E.

An expression of the form @E returns a pointer to the 8 bit or 32 bit memory location specified by E. E must be a variable name or an expression with leading operator ! or %.

Expressions of the form: +E, -E, ~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, ~ returns the bitwise complement of the value, ABS returns the absolute value, and NOT returns the boolean complement of the value.

3.6 Infixed Expression Operators

An expression of the form E1!E2 evaluates E1 and E2 to yield respectively a pointer, *p* say, and an integer, *n* say. The value returned is the 32 bit contents of the n^{th} word relative to *p*.

An expression of the form E1%E2 evaluates E1 and E2 to yield a pointer, *p* say, and an integer, *n* say. The expression returns a 32 bit unsigned result equal to the byte at position *n* relative to *p*.

An expressions of the form E1<<E2 (or E1>>E2) evaluates E1 and E2 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.

Expressions of the form: E1*E2, E1/E2, E1 MOD E2, E1&E2, E1 XOR E2, return the result of applying the given operator to the values of 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 relop E relop ... relop E where each relop is one of =, ~=, <=, >=, < or > returns TRUE if all the individual relations are satisfied and FALSE, otherwise. The operands are evaluated from left to right, and evaluation stops as soon as the result can be determined. No operand is evaluated more than once.

An expressions of the form: E1 AND E2 or E1 OR E2 returns the boolean value obtained by applying the given operator to the boolean values of E1 and E2. If the result can be determined from E1 alone, then E2 is not evaluated.

An expression of the form: E1->E2,E3 first evaluates E1, and, if this yields FALSE, it returns the value of E3, otherwise it return the value of E2.

3.7 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).

3.8 MATCH and EVERY Expressions

A MATCH expression has the following form:

```
MATCH args
: P ,.., P => Clist
  ...
: P ,.., P => Clist
```

It consists of the word MATCH followed by an argument list, followed by zero or more match items (described in Section 5). The argument list is 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. 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 items are 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.

3.9 Expression Precedence

A lexical token that can start an expression or a pattern cannot denote an infixed or postfixed operator when occuring as the first token of a line. This rule applies to the tokens: !, %, +, ++, +++, -, --, ---, =, ~=, <=, >=, < and >, and its purpose is to allow most semicolons in command sequences to be omitted.

Table 1 specifies the precedence of the various expression constructs. The prcedence values are in the range 0 to 14, 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).

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

Table 1

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
     NOT x=y    means  NOT (x=y)
```

3.10 Manifest Constant Expressions

A manifest constant expression is an expression that can be evaluated at compile time to yield an integer. It may only contain 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.

3.11 Static Constant Expressions

A static constant expression may be used to specify 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 constructs

[SK0,..,SKn], VEC K and CVEC K, where SKi and K denote static constant expressions and manifest constant expressions, respectively, are allowed and create static vectors.

4 Commands

The primary purpose of commands is for their side effects on the values of 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 the 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 3.6) leads to a memory location which is the one that is updated by the assignment. If the % operator is used then the appropriate 8 bit location is updated by the least significant 8 bits of the value of E.

A simultaneous assignment has the following form:

```
L1,..,Ln := E1,..,En
```

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.

```
L1,..,Ln ALL:= E
```

The following assignments

```
LET V1,..,Vn = E1,..,En
LET V1,..,Vn ALL= E
```

where `V1,..,Vn` are variable names, cause dynamic locations for the variables to be allocated when control enters the current dynamic scope (see Section 7). They are otherwise equivalent to the assignments:

```
V1,..,Vn := E1,..,En
V1,..,Vn ALL:= E
```

respectively.

An assignment of the form:

```
L1,..,Ln op:= E1,..,En
```

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.

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 `C1` and `C2` denote commands. To execute a conditional command the expression `E` is first evaluated. If it yields a non zero value and `C1` is present then `C1` is executed. If it yields zero and `C2` is present, `C2` 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 `WHILE` command repeated executes the command `C` so long as `E` yields a non zero value. 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, respectively, like `WHILE E DO C` or `UNTIL E DO C`.

The `FOR` command first initialises its control variable (`vid`) to the value of `E1`, and evaluates the end limit specified by `E2`. 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 3.10). 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 in 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 command 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 functions.

The GOTO command takes an argument list which is syntactically similar to the argument list of a function call (see Section 3.3). 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 function call 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 passes 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 fragment of program.

```
MANIFEST
Id=1, 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)
: ?,               ? => RAISE Eval

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

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. Syntactically, the semicolon is only needed if there is any ambiguity about where one command ends and the next begins. The semicolon can always be omitted if the second command starts on a new line. For this to work, expression and pattern operators which can be both prefixed operators and infixed or postfixed may only occur as the first token on a line with its prefixed meaning (see Section 3.9). Remember also from Section 3.3 that the start of a function argument list must be on the same line as the end 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 decementing 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. Each match item has the following syntactic form:

```
: Plist => Clist
```

where `Plist` is a list of zero or more patterns separated by commas and `Clist` is a sequence of commands, optionally separated by semicolons. At the moment control is passes 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 given 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 equalls 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 succesfully, the assignment is performed with the implied left hand operand, just prior to executing the corresponding command sequence.

If two patterns are justaposed, 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, 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 four kind 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 `void`.
`p`	A pointer type.
`s`	The type `short`.
`i`	The type `int`.
`u`	The type `unsigned int`.
`l`	The type `long int`.
`_`	This can occur as the last character of a qualifier to indicate that the function takes a variable number of arguments, compatible with the ... 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 N1=K1 ,..., Nn=Kn
```

where `N1,..,Nn` are constant identifiers (see Section 3.1) and `K1,..,Kn` are manifest constant expressions (see Section 3.10). 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 (`=Ki`) is omitted, the a value one larger than the previously defined manifest constant is implied, and if `=K1` 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 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 3.11). 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 given by default to be `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"`.

6.4 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). 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 is passed 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.

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

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. The scope of a MANIFEST identifier starts from the place that it is declared and extends to the end of the program. It is thus necessary to declare manifests before they are used. 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 right 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. The scope of a static identifier extends from its declaration to the end of the program, and so, as with manifests, such variables must be declared before they are used.

All other variables are dynamic, being allocated space within the stack frame of a function. The scope of such a variable is the *dynamic scope* associated with the place in the program where the variable is declared. Space is allocated for a variable when control passes into its dynamic scope, and it is freed when control leaves the scope.

A dynamic scope is the 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. is the body of a VALOF expression, or
4. extends from the word FOR to then end of the body of the FOR command.

Thus, the scope of an indentifier 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 =>, VALOF or FOR symbol.

Dynamic scopes also determine the dynamic lifetime (or extent) of local vectors declared by VEC, CVEC or [E,..,E]. Space for such 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 Example Programs

9.1 Coins

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

```
GET "mcpl.h"

FUN ways
: 0,              ?  => 1
: ?,            [0]  => 0
: sum, coins[>sum]  => ways(sum, @ coins!1)
: sum, coins[ val]  => ways(sum, @ coins!1) + ways(sum-val, coins)

FUN t : sum =>
    printf("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 100
```

9.2 Primes

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

```
GET "mcpl.h"

MANIFEST Upb = 999

FUN start : =>
  printf "\nTable of prime numbers\n\n"
  LET count, isprime = 0, 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, i := FALSE, i+p
    printf(" %i3", n)
    IF ++count MOD 10 = 0 DO newline()
  }
  printf "\nEnd of output\n"
```

9.3 Queens

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

```
GET "mcpl.h"

STATIC count = 0

FUN try
:  ?, #xFF,  ? => count++
: ld,  row, rd => LET poss = ~(ld | row | rd) & #xFF
                  UNTIL poss=0 DO
                  { LET bit = poss & -poss
                    poss -:= bit
                    try( (ld|bit)<<1, row|bit, (rd|bit)>>1 )
                  }
FUN start
: => try(0, 0, 0)
     printf("\nNumber of solutions is %d\n", count)
```

9.4 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 febdays : year => year MOD 400 = 0 -> 29,
                      year MOD 100 = 0 -> 28,
                      year MOD 4   = 0 -> 29,
                      28
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
      printf(" %3d %ss\n", count!day, dayname!day)
```

9.5 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 tautologies. The program includes both a parser for the expression and a prover function pr to test whether it is always true.

```
GET "mcpl.h"

MANIFEST
Id, Not, And, Or,       // Expression node operators
Syntax=100              // Syntax exception

STATIC
str, strp, ch1, ch2, ch3, token, spacep

FUN member : ?,         0 => FALSE
           : x, [ ?, =x] => TRUE
           : x, [ys,  ?] => member(x, ys)

FUN add : x, xs => member(x, xs) -> xs, mk2(x, xs)

FUN pr
:  ?,  0,  ?,               0 => FALSE

: al,  0, ar,    [[Not,x], cr] => pr(al, [x,0], ar, cr)

: al,  0, ar, [[And,x,y], cr] => pr(al, 0, ar, [x,cr])
                                 AND
                                 pr(al, 0, ar, [y,cr])

: al,  0, ar,  [[Or,x,y], cr] => pr(al, 0, ar, [x,[y,cr]], x))

: al,  0, ar,     [[Id,x], cr] => member(x, al)
                                 OR
                                 pr(al, 0, add(x,ar), cr)

: al,   [[Not,x],cl], ar, cr  => pr(al, cl, ar, [x,cr])
: al, [[And,x,y],cl], ar, cr  => pr(al, [x,[y,cl]], ar, cr)

: al,  [[Or,x,y],cl], ar, cr  => pr(al, [x,cl], ar, cr)
                                 AND
                                 pr(al, [y,cl], ar, cr)

: al,    [[Id,x],cl], ar, cr  => member(x,ar)
                                 OR
                                 pr(add(x,al), cl, ar, cr)

// A ,.., Z  -->  [Id, 'A'] ,.., [Id, 'Z']
// ~x        -->  [Not, x]
// x & y     -->  [And,x,y]
// x | y     -->  [Or,x,y]
// x -> y    -->  ~x | y
// x <-> y   -->  (x -> y) & (y -> x)

FUN rch : => ch1, ch2, ch3 := ch2, ch3, str%strp
             UNLESS ch3=0 DO strp++

FUN parse : s => str, strp := s, 0
                 rch(); rch(); rch()
                 nexp 0

FUN lex : => MATCH (ch1, ch2, ch3)
: ' ' | '\n'    => rch(); lex()
: ( 'A'..'Z' | '(' | ')' | '~' | '&' | '|' )
  ch             => rch(); ch
: '-', '>'       => rch(); rch(); '->'
: '<', '-', '>'  => rch(); rch(); rch(); '<->'
:                => RAISE Syntax

FUN prim : => MATCH token
: 'A'..'Z' => LET a = mk2(Id, token)
              lex()
              a
: '('      => LET a = nexp 0
              UNLESS token=')' RAISE Syntax
              lex()
              a
: '~'      => mk2(Not, nexp 3)
:          => RAISE Syntax

FUN nexp : n => lex(); exp n
FUN exp : n =>
    LET a = prim()
    { MATCH (token, n)
       :  '&', <3 => a := mk3(And, a, nexp 3)
       :  '|', <2 => a := mk3(Or , a, nexp 2)
       : '->', <1 => a := mk3(Or, mk2(Not, a), nexp 1)
       :'<->', <1 => LET b = nexp 1
                     a := mk3(And, mk3(Or, mk2(Not, a), b),
                                   mk3(Or, mk2(Not, b), a)
                             )
       :          => RETURN a
    } REPEAT

FUN try : e =>
    spacep := @ (VEC 10000)!10000
    { TEST pr(0, 0, 0, [parse e, 0])
      THEN printf("%s is TRUE\n", e)
      ELSE printf("%s is FALSE\n", e)
    } HANDLE : Syntax => printf "Bad syntax\n"

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

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

FUN start : =>
  printf "associative laws of & and | \n"
  try "(P & Q) & R  <->  P & (Q & R)"
  try "(P | Q) | R  <->  P | (Q | R)"

  printf "distributive laws of & and | \n"
  try "(P & Q) | R  <-> (P | R) & (Q | R)"
  try "(P | Q) & R  <-> (P & R) | (Q & R)"

  printf "Laws 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)"

  printf "Classical 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.
*/

printf "Problem 5 \n"
try "((P|Q)->(P|R)) -> (P|(Q->R))"

printf "Problem 9 \n"
try "((P|Q) & ( ~P | Q) & (P | ~Q)) ->  ~( ~P | ~Q)"

printf "Problem 12.  Dijkstra's law \n"
try "((P <-> Q) <-> R)  ->  (P <-> (Q <-> R))"

printf "Problem 17"
try "(P & (Q->R) --> S)  <->  ((~P|Q|S) & (~P|~R|S))"

printf "False 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)"

printf "Indicates need for subsumption \n"
try "((P & (Q<->R))<->S)  <->  ((~P|Q|S) & ( ~P|~R|S))"
```

9.6 Eval

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

```
GET "mcpl.h"

MANIFEST Id=1, Num, Times, Div, Pos, Neg, Plus, Minus,
         Eq, Cond, Lam, Ap,
         Syntax=1, Lookup, Eval

STATIC str, strp, ch, nch, token, 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))
                        }
: ?,             ? => RAISE Eval

// Construct         Corresponding Tree

// a ,.., z    -->   [Id, 'a'] ,.., [Id, 'z']
// dddd        -->   [Num, dddd]
// x y         -->   [Ap, x, y]
// 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]
```

```
FUN rch : => ch := nch
             nch := str%strp++
             UNLESS nch=0 DO strp++

FUN parse : s => str, strp := s, 0
                 rch(); rch(); nexp 0

FUN lex : => MATCH (ch, nch)
: ' '|'\n', ?  => rch(); lex()
: '(' | ')' | '*' | '/' | '+' | '-' | 'L' | '.',
            ? => 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
: '-',    '>' => token := '->'; rch()
:   ?,     ?  => RAISE Syntax

FUN prim : => MATCH token
: Id   => LET a = mk2(Id, lexval)
          lex()
          a
: Num  => LET a = mk2(Num, lexval)
          lex()
          a
: '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,
                ? => a := mk3(   Ap, a, nexp 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(b)
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 => printf("%s\n", s)
FUN wrn : n => printf("%d\n", n)

FUN try : e => wrs e
               spacep := @ (VEC 10000)!10000

               wrn ( eval(parse e, 0) )

               HANDLE : Syntax => printf "Bad syntax"
                      : Lookup => printf "Bad lookup"
                      : Eval   => printf "Bad eval"

FUN start : =>
    try "1=2 -> 1234, 3*4+100"        // Answer 112
    try "(Lx x+1) ((Lx x) (Ly y) 99)" // Answer 100
    try "(Ls Lk s k k)            \
         \  (Lf Lg Lx f x (g x)) \
         \     (Lx Ly x)          \
         \        (Lx x)          \
         \           99"          // Answer 99
```

9.7 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 Nth 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,
   MSB     = N>>1,
   LSB     = 1

STATIC v = VEC Upb, w = VEC Upb

FUN start : =>
   FOR i = 0 TO Upb DO v!i := i

   pr(v, 15)
// Prints the original data
//     0      1      2      3      4      5      6      7
//     8      9     10     11     12     13     14     15

   dofft v

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

   invfft v

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

```
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, b1 = a>>1, b<<1
                    reorder(@p!a, @q!b, a1, b1)
                    reorder(p,    q,    a1, b1)

FUN fft : nn, v, pp, msb =>
    LET n, p = nn>>1, 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 => FOR i = 0 TO upb DO
                   { printf("%5d ", v!i)
                     IF i MOD 8 = 7 DO printf "\n"
                   }
                   printf "\n"
```

```
FUN dv
: 1, m,    ?, ? => m
: 0, m,    ?, n => m-n
: a, m, b >a, 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)
```

9.8 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 represented by the six `1`s.

```
GET "mcpl.h"

MANIFEST Upb = 5000

STATIC ltape, rtape, spacep

FUN prb
:          0 => RETURN
:    [0, ch] => wrch ch
:  [chs, ch] => prb chs; wrch ' '; wrch ch
```

```
FUN prf
:          0 => RETURN
:    [0, ch] => wrch ch
:  [chs, ch] => wrch ch; wrch ' '; prf chs

FUN pr : x, c, y  => prb x; printf("[%c]", c); prf y; newline()

FUN right : c => MATCH rtape
:           0 => ltape := mk2(ltape, c)
: [link, ch] => LET oldch = ch
                ltape,rtape,link,ch := rtape,link,ltape,c
                oldch

FUN left : c => MATCH ltape
:           0 => rtape := mk2(rtape, c)
: [link, ch] => LET oldch = ch
                rtape,ltape,link,ch := ltape,link,rtape,c
                oldch

FUN halt : c => pr(ltape, c, rtape)

FUN turing : lstr, ch, rstr =>
    LET i = 0
    ltape, rtape ALL:= 0
    UNTIL lstr%i=0 DO i++
    UNTIL i=0      DO ltape := mk2(ltape, lstr%--i)
    UNTIL rstr%i=0 DO rtape := mk2(rtape, rstr%i++)
    s0 ch

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 )
```

```
FUN mk2 : x, y => !---spacep := y
                  !---spacep := x
                  spacep

FUN start : => spacep := @ (VEC Upb)!Upb
               printf "Turing entered\n"
               turing("A11", "B", "111A")
```

10 References

[1] Richards,M. and Whitby-Strevens, C. *BCPL: the Language and its Compiler*, Cambridge University Press, 1979.

[2] Paulson, L.C., *ML for the Working Programmer*, Cambridge University Press, 1991.

[3] Harbison, S.P. and Steele,G.L., *C – A Reference Manual*, Prentice-Hall, 1987.

[4] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*, Springer-Verlag, 1981.