HCMC University of Technology
Faculty of Computer Science & Engineering

MP

Micro Pascal Language

Author

Dr. Nguyen Hua Phung

June 2017

Contents

MP

version 1.2

1 Introduction

MP (Mini Pascal) is a language which consists of a subset of Pascal plus some Java language features.
The Pascal features of this language are (details will be discussed later): a few primitive types, one-dimensional arrays, control structures, expressions, compound statements (i.e., blocks), functions and procedures.
The Java features of this language are as follows:

1. MP has multiple assignments, borrowed from Java.
2. In MP, like Java, the operands of an operator are guaranteed to be evaluated in a specific evaluation order, particularly, from left to right. In Pascal, the evaluation order is left unspecified. In the case of *f() + g()*, for example, MP dictates that *f* is always evaluated before *g*.

For simplicity reason:

1. There is only one dimension array in MP.

```
var i:array [1..2,3..4] of integer;       // ERROR
var i:array [1..5] of integer;   // CORRECT
```

Conventionally, the **sequence '\n' must be used as a new line character in MP**.

2 Program Structure

MP does not support separate compilation so all declarations (variable and function) must be resided in one single file.

An MP program consists of many declarations which can be a variable or function or procedure declaration.

2.1 Variable declaration:

A variable declaration starts with keyword **var** and then a list of declarations each of which starts with a comma-separated list of identifiers, a colon (:), a type and ends with a semicolon.

For example,

```
var a,b,c:integer;
        d: array [1..5] of integer;
        e,f: real;
```

2.2 Function declaration:

In MP, a function declaration specifies the name of the function, the type of the return value and the number and types of the arguments that must be supplied in a call to the function as well as the body of the function. A function is declared as follows:
function *<function-name> '(' <parameter-list> ')' ':' <return type> ';' <variable declaration> <compound-statement>*
where

- ***function*** is a keyword
- *<function-name>* is an identifier used to represent the name of the function.
- *<parameter-list>* is zero or more *<parameter-declaration>*'s separated by ';'. A *<parameter-declaration>* is declared as follows:
  <comma-separated list of identifier> ':' <type>
- *<type>* is the function return type.
- *<variable declaration>* is described in the previous section. The variable declaration may be ignored.
- *<compound-statement>* is described in Section 6.8

For example,

```
function foo(a,b:integer;c:real):array [1..2] of integer;
var x,y: real;
begin
  ...
end
```

2.3 Procedure declaration:

The procedure declaration is like a function one except that the keyword ***procedure*** is used instead the keyword ***function*** and there is no colon and the return type after the parameter declaration part.

For example,

```
procedure foo(a,b:integer;c:real);
var x,y: real;
begin
  ...
end
```

MP does not support function/procedure overloading. Thus, a function must be defined exactly once.

MP does not support nested function/procedure as in Pascal. For example: the following declaration is invalid in MP

```
function foo(i:integer):real;
        procedure child_of_foo(real f)
         begin
        ...
        end
begin
end//ERROR
```

3 Lexical Specification

This section describes the character set, comment conventions and token set in the language.

3.1 Character Set

An MP program is a sequence of characters from the ASCII character set. Blank, tab, formfeed (i.e., the ASCII FF), carriage return (i.e., the ASCII CR) and newline (i.e., the ASCII LF) are *whitespace characters*. A line is a sequence of characters that ends up with a LF. This definition of lines can be used to determine the line numbers produced by an M compiler.

3.2 Comments

There are three kinds of comments:

- A traditional block comment:
  (This is*
  *a block comment *)*
  All the text from (* to *) is ignored.

- A block comment:
  { This is a block comment }
  All the text from { to } is ignored.

- A line comment:
  //This is a line comment
  All the text from // to the end of the line is ignored.
  As designed in C, C++ and Java, the following rules are also enforced:

  – Comments do not nest.
  – (*, *), { and } have no special meaning in comments that begin with //.
  – // has no special meaning in comments that begin with (* or {.

3.3 Token Set

In an MP program, there are five categories of tokens: identifiers, keywords, operators, separators and literals.

- Identifiers: An identifier is an unlimited-length sequence of letters, digits and underscores, the first of which must be a letter or underscore. MP is **case-insensitive**, meaning that *abc* and *Abc* are the same.

- Keywords: The following character sequences are reserved as *keywords* and cannot be used as identifiers:
  break continue for to downto do if then else return while begin end function procedure var true false array of real boolean integer string not and or div mod
  Some keywords are used as operators, or literals. These keywords are also **case-insensitive**.

- Operators: The following 15 *operators* in MP:

Operator	Meaning	Operator	Meaning
+	Addition	-	Subtraction or negation
*	Multiplication	/	Division
not	Logical NOT	mod	Modulus
or	Logical OR	and	Logical AND
<>	Not equal	=	Equal
<	Less than	>	Greater than
<=	Less than or equal	>=	Greater than or equal
div	Integer division		

3.4 Separators

The following characters are the **separators**: left square bracket ('['), right square bracket (']'), colon (':'), left bracket ('('), right bracket (')'), semicolon (';'), double dot ('..') and comma (',').

3.5 Literals

A **literal** is a source representation of a value of either an integer type, real type, boolean type or string type.

- An *integer literal* is **always expressed in decimal (base 10)**, consisting of a sequence of at least one digit. An integer literal is of type **integer**.

- A *floating-point literal* has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part and an exponent. The exponent, if present, is indicated by the ASCII letter **e** or **E** followed by an optionally signed (-) integer. At least one digit, in either the whole number or the fraction part, and either a decimal point or an exponent are required. All other parts are optional. A floating-point literal is of type **real**.
  For example: The following are valid floating literals:
  1.2 1. .1 1e2 1.2E-2 1.2e-2 .1E2 9.0 12e8 0.33E-3 128e-42
  The following are **not** considered as floating literals:
  e-12 (no digit before 'e') 143e (no digits after 'e')

- The *boolean literal* has two values, represented by the literals **true** and **false**, formed from ASCII letters.

- A *string literal* consists of zero or more characters enclosed in double quotes '"'. The quotes are not part of the string, but serve to delimit it.
  It is a compile-time error for a backspace, newline, formfeed, carriage return, tab, single quote, double quote or a backslash to appear after the opening '"' and before

the closing matching '"'. The following escape sequences are used instead:

\b	backspace
\f	formfeed
\r	carriage return
\n	newline
\t	horizontal tab
\'	single quote
\"	double quote
\\	backslash

A string literal is of type **string**.

4 Types and Values

Types of all variables and expressions in MP must be known at compile time. Types limit the values that a variable can hold (e.g., an identifier x has type integer cannot hold value true...), the values that an expression can produce, and the operations supported on those values (e.g., we cannot apply a plus operator to 2 boolean values...).
MP types are divided into two categories:

- Primitive types: **boolean, integer, real, string**.
- Compound type: **array**.

4.1 The boolean Type and Values

The **boolean** type represents a logical quantity with two possible values: *true* and *false*. The following operators can act on boolean values:
not, and, and then, or, or else
A *boolean expression* is an expression that evaluates to *true* or *false*. Boolean expressions determine the control flow in **if**, **for** and **while**. Only boolean expressions can be used in these control flow statements.

While the first operator (not) is unary, the others are binary. The operators *and* and *or* are evaluated normally which means their operands are calculated before these operators. Meanwhile, the operators *and then* and *or else* are short-circuited evaluated which means that the left operand is evaluated firstly and the right one is evaluated only when necessary.

4.2 The integer Type and Values

The values of type **integer** are 32-bit signed integers in the following ranges:
-2147483648 ... 2147483647
The following operators can act on integer values:
+ - * div mod < <= > >= <> = /
The first five operators always produce a value of type **integer**. The next six operators always result a value of type **boolean**. The last operator (/) will result a value of type **real** when both operands are in type **integer**.
Here, - represents both the binary subtraction and unary negation operators.

4.3 The real Type and Values

A real value is a 32 bit single-precision number. The exact values of this type are implementation-dependent. The following operators can act on floating-point values:

- The binary arithmetic operators +, -, * and /, which result in a value of type **real**.
- The unary negation operators -, which results in a value of type **real**.
- The relational operators =, <>, <, <=, > and >=, which result in a value of type **boolean**.

4.4 The string Type and Values

Strings can only be used in an assignment or passed as a parameter of a function invocation. For example, a string can be passed as a parameter to the built-in function *putString()* or *putStringLn()* as described in Section 7.

4.5 Array Types and Their Values

MP supports only **one-dimensional arrays**. Originally, arrays in Pascal support the following features:

- The lower bound and the upper bound of the index must be provided in an array declaration.
- A subscript can be any integer expression, i.e., any expression of type **integer**.

However, for simplicity purpose, one-dimensional arrays in MP are more restrictive :

- The element type of an array can only be a primitive type such as **boolean**, **integer**, **real** or **string**.
- An array variable itself (without the associated square brackets []) can be only used as a actual parameter to pass to/from a function or procedure.

5 Expressions

An expression is a finite combination of operands and operators. An operand of an expression can be a literal, an identifier, an element of an array or a function call.

5.1 Precedence and Associativity

The rules for precedence and associativity of operators are shown as follows:

Operator	Arity	Notation	Precedence	Associativity
- not	unary	prefix	Highest	right to left
/ * div mod and	binary	infix		left to right
+ - or	binary	infix		left to right
= <> < <= > >=	binary	infix		none
and then, or else	binary	infix	Lowest	left to right

The operators on the same row have the same precedence and the rows are in order of decreasing precedence. An expression which is in '(' and ')' has highest precedence.

5.2 Type Coercions

In MP, like C and Java, mixed-mode expressions whose operands have different types are permitted.

The operands of the following operators:
+ - * / < <= > >= = <>
can have either type **integer** or **real**. If one operand is **real**, the compiler will implicitly convert the other to **real**. Therefore, if at least one of the operands of the above binary operators is of type **real**, then the operation is a floating-point operation.

The following type coercion rules for an implicit or explicit assignment are permitted:

- If the type of the LHS is **real**, the expression in RHS must have either the type **integer** or **real** or a compile-time error occurs.

5.3 Index Expression

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:
<expression> '[' expression ']'
The type of the first *<expression>* must be an array type. The second expression, i.e. the one between '[' and ']', must be of integer type. The index operator returns the corresponding element of the array.
For example,
foo(2)[3+x] := a[b[2]] +3;
The above assignment is valid if variables a, b and the return type of function foo are in an array type, x is in **integer** type and the element type of array b is **integer**.

5.4 Invocation Expression

An invocation expression is a function call which starts with an identifier followed by "(" and ")". A nullable comma-separated list of expressions might be appeared between "(" and ")" as a list of arguments.

Like C, all arguments (including arrays) in MP are passed "by value." The called function is given its value in its parameters. Thus, the called function cannot alter the variable in the calling function in any way.

When a function is invoked, each of its parameters is initialized with the corresponding argument's value passed from the caller function.

When an array variable is passed (as an argument) to/from a function/procedure, the lower bound, upper bound and the element type of the array argument and the array formal parameter must be the same. All members of the array argument will be copied to the corresponding members of the array formal parameter.

The type coercion rules for assignment are applied to parameter passing where LHS's are formal parameters and RHS's are arguments.

For example,

```
procedure foo(a:array [1..2] of real) ...
procedure goo(x:array [1..2] of real);
    var
        y: array [2 .. 3] of real;
        z: array [1 .. 2] of integer;
    begin
        foo(x); //CORRECT
        foo(y); //WRONG
        foo(z); //WRONG
    end
```

The type coercion rules and the exception in parameter passing are also applied to return type where LHS is the return type and RHS is the expression in the return statement.

For example,

```
function foo(): real;
    begin
        if (...) then return 2.3; //CORRECT
        else return 2; //CORRECT
    end
```

and,

```
function foo(b:array [1..2] of integer):array [2 .. 3] of real;
```

```
var
    a: array [2 .. 3] of real;
begin
    if () then return a; //CORRECT
    else return b; //WRONG
end
```

5.5 Evaluation Order

MP requires the left-hand operand of a binary operator must be evaluated first before any part of the right-hand operand is evaluated.

Similar, in a function call (called a method call in Java), the actual parameters must be evaluated from left to right.

Every operand of an operator must be evaluated before any part of the operation is performed. The two exceptions are the logical operators *and then* and *or else*, which are still evaluated from left to right, but it is guaranteed that evaluation will stop as soon as the truth and falsehood is known. This is known as the **short-circuit evaluation**.

6 Statements and Control Flow

MP supports these statements: **assignment**, **if**, **for**, **while**, **break**, **continue**, **return**, **call**, **compound**, and **with**. All statements except **if**, **for**, textwhile, **compound** and the **with** one must be followed by a semi-colon.

6.1 Assignment Statement

Like Java, the assignment statement may have many left hand sides but just one right hand side. The assignment statement may be written as follows:

lhs_1 := lhs_2 := ... lhs_n := <*expression*>

The left hand sides (lhs_1,...,lhs_n) can be a scalar variable or an index expression.

For example,

```
a := b[10] := foo()[3] := x := 1 ;
```

The <*expression*> is calculated first and its value is assigned to lhs_n. Then the value of lhs_n is assigned to lhs_{n-1} and so on. The type coercion rule is applied to each assignment.

6.2 The if Statement

There are two types of **if** statement: if-else and if-no else. The if-else is written as follows:
if <*expression*> *then*

<*statement1*>

else
    <statement2>
where *<expression>*, which must be of the type **boolean**, is first evaluated. If it is *true*, *<statement1>* is executed. The *<statement2>* is otherwise executed.

The if-no else is like if-else but there is no *else* and *<statement2>*. In this type of if statement, if the *<expression>* is *false*, the next statement will be executed.

Like C, C++ and Java, the MP language suffers from the so-called dangling-else problem. MP solve this by decreeing that an else must belong to the innermost if.

6.3 The while Statement

while <expression> do <statement>

When **while** statement is executed, the *<expression>* is evaluated first. While its value is *true*, the *<statement>* is executed. The loop stops when the *<expression>* is *false*.

6.4 The for Statement

The **for** statement is written as follows:
for <identifier> := <expression$_1$> (to/downto) <expression$_2$> do <statement>

where *<expression$_1$>* is executed first and its value is assigned to *<identifier>*. If the keyword *to* is used and the value of the *<identifier>* is less than or equal to the value of *<expression$_2$>*, the *<statement>* is executed and then the value of *<identifier>* is increased by one. The comparion between *<identifier>* and *<expression$_2$>*, the execution of *<statement>*, and the increase of *<identifier>* are repeated until the result of the comparison becomes false. If the keyword *<downto>* is used, the process happened the same except that the relational operator is greater than or equal to and the increase of *<identfier>* is replaced by the decrease.

The *<identifie>* must be a local integer variable.

6.5 The break Statement

This statement must appear inside a loop such as **for** or **while**. When it is executed, the control will transfer to the statement next to the enclosed loop. This statement is written as follows:
break ';'

6.6 The continue Statement

This statement must appear inside a loop such as **for** or **do while**. When it is executed, the control will jump to the end of the body of the loop. This statement is written as

follows:
continue ';'

6.7 The return Statement

A **return** statement aims at transferring control to the caller of the function/procedure that contains it.

A **return** statement with no expression must be contained within a procedure while the **return** with expression must be in a function.

6.8 The compound Statement

A **compound** statement starts with keyword **begin** and ends with keyword **end**. Between these two keywords, there is a nullable list of statement.

6.9 The with statement

A **with** statement is written in a form:
with *<list of variable declaration>* ***do*** *<statement>*

For example,

```
with a,b:integer;c:array [1..2] of real; do
        d = c[a] + b;
```

6.10 Call statement

A call statement starts with a *<identifier>*, which is a procedure name, followed by a nullable comma-separated list of expressions enclosed by round brackets. For example,

```
foo(3, a+1, m(2));
```

7 Built-in Functions

MP has some following built-in functions:
function getInt():integer: reads and returns an integer value from the standard input
procedure putInt(i:integer): prints the value of the integer i to the standard output
procedure putIntLn(i:integer): same as putInt except that it also prints a newline
function getFloat():real: reads and returns a floating-point value from the standard input
procedure putFloat(f:real): prints the value of the real f to the standard output
procedure putFloatLn(f:real): same as putFloat except that it also prints a newline
procedure putBool(b:boolean): prints the value of the boolean b to the standard output
procedure putBoolLn(b:boolean): same as putBoolLn except that it also prints a new line

procedure putString(s:string): prints the value of the string to the standard output
procedure putStringLn(s:string): same as putStringLn except that it also prints a new line
procedure putLn(): prints a newline to the standard output

8 Scope Rules

Scope rules govern declarations (defining occurrences of identifiers) and their uses (i.e., applied occurrences of identifiers).

The scope of a declaration is the region of the program over which the declaration can be referred to. A declaration is said to be in scope at a point in the program if its scope includes that point.

There are three levels of scope: global, function/procedure, and block.

A global scope is the whole program which is applied to all function/procedure declarations and variable declarations outside function/procedure declarations. All built-in function/procedures are applied to this scope

A function/procedure scope is the entire corresponding function/procedure which is applied to its parameters and variable declarations just after the parameter part.

A block scope is the statement inside the *with* statement. The scope is applied to the declarations between the keywords **with** and **do**.

There are four additional rules on the scope restrictions:

1. All declarations in global scope are effective in the entire progran.
2. All declarations in local scope are effective from the place of the declaration to the end of its scope.
3. No identifier can be defined more than once in the same scope. This implies that **no** identifier represents both a global variable and a function name simultaneously.
4. *Most closed nested rule*: For every applied occurrence of an identifier in a block, there must be a corresponding declaration, which is in the smallest enclosing block that contains any declaration of that identifier.

Consider the following MP program:

```
1 var i:integer;
2 function f():integer;
3 begin
4     return 200;
5 end
6 procedure main();
```

```
 7 var
 8     main: integer;
 9 begin
10     main := f();
11     putIntLn(main);
12     with
13         i:integer;
14         main:integer;
15         f:integer;
16     do begin
17         main := f := i := 100;
18         putIntLn(i);
19         putIntLn(main);
20         putIntLn(f);
21     end
22     putIntLn(main);
23 end
24 var g:real;
```

The above program will be compiled and print the following results:

```
200
100
100
100
200
```

In this program, there are three scope levels:

Declaration	Level	Scope
putIntLn (built-in)	1	Entire program
i (line 1)	1	Entire program
f (line 2)	1	Entire program
main (line 6)	1	Entire program
main (line 8)	2	line 9-12 and line 22-23
i (line 13)	3	line 13-21
main (line 14)	3	line 13-21
f (line 15)	3	line 13-21
g (line 24)	1	Entire program

Note that the variable g declared in line 24 has the global scope although it is declared at the end of the program.

The variable *main* declared in line 8 is said to *hide* the procedure declaration *main* in line 6. The variable *main* declared in line 14 hides the variable declaration *main* in line 8. The variable i declared in line 13 hides the global variable i in line 1. The variable f

declared in line 15 hides the function declaration f in line 2.

The scopes of the declarations f in line 2, i in line 1 and main in line 6 are not contiguous. Such gaps are known as scope holes, where the corresponding declarations are hidden or invisible. As a matter of style, it is advised not to introduce variables that conceal names in an outer scope. This is the major reason why Java disallows a variable declaration from hiding another variable declaration of the same name in an outer scope. Therefore, the MP program above is a bad programming style.

9 The main function

A special procedure, i.e. main procedure, is an entry of a MP program where the program starts:

```
procedure main(); // no parameters are allowed
begin
  ...
end
```

10 Change Log

- Different from version 1.1
  - Add a sentence "The variable declaration may be ignored." in Section 2.2
  - Fix the reference to Section 6.8 in Section 2.2
  - Add a sentence "These keywords are also **case-insensitive**." in the description of **Keywords** in Section 3.3.
  - Fix the example in Section 9