MBL:
A Language for Teaching Compiler Construction

John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, CANADA T2N 1N4

aycock@cpsc.ucalgary.ca

This paper presents a specification for MBL, an imperative programming language designed to help teach compiler construction. A minimalist approach was taken to MBL's design, adding language constructs only if they provided distinct challenges in implementation. In addition, some anomalies were deliberately retained to stress particular areas of compiler design and implementation.

Research Report 95/574/26

Introduction

MBL (pronounced ''mumble'') is an imperative language designed as a project for a two-semester compiler construction course. The intention was to create a small language, choosing language elements based on the skills and techniques required to implement them. An attempt has been made to keep the language independent of the operating environment and target machine.

Unfortunately for the MBL user, this document is written as a reference for compiler writers, not as a user manual. The reader is assumed to be familiar with EBNF grammar notation[1] , at least one imperative language, and should be prepared to reread sections in order to capture all interdependencies.

This document has benefited from the comments of several proofreaders: Cliff Marcellus read an early draft, and Shannon Jaeger repeatedly deciphered both handwritten and typeset copies. Anton Colijn's comments helped pinpoint some remaining loopholes.

> Some sections are followed by a description of relevant details of the MBL "reference implementation", inset in boxes such as this one.
>
> The reference implementation is a compiler for MBL which runs under UNIX™ and outputs assembly code for the MIPS® R2000 architecture[2,3] . It and the MBL run-time system are being written in C[4] and together should be approximately 4500 lines of code. YACC[5,6] is used for the parser, but all other parts are hand-coded.

UNIX is a trademark.
MIPS is a registered trademark of MIPS Computer Systems, Inc.

Lexical Elements

Whitespace

Whitespace characters include spaces, horizontal tabs, newlines, and possibly other implementation-defined characters. It must be used to separate adjacent identifiers, and may be used freely to separate other lexical elements.

With the exception of string and character literals, whitespace may not appear inside a lexical element.

Numbers

Numbers consist of a digit followed by zero or more digits; digits are in base ten. Numbers differ from integers in that a number is simply a sequence of digits, whereas an integer is a built-in data type with minimum and maximum values.

Negative numbers are formed by applying the unary minus to a number, making them constant expressions rather than single lexical elements. So "`-123`" contains *two* lexical elements: the subtraction sign ("`-`") and a number ("`123`").

Identifiers

Identifiers consist of an alphabetic character followed by zero or more alphanumeric characters. All identifiers are case insensitive, meaning that corresponding upper- and lower-case letters are treated as equivalent.

Certain identifiers are reserved and may not be redefined by the user (these are also known as "reserved words"):

```
and           constant   if       of
array         else       import   or
associative   end        in       procedure
break         export     list     record
case          for        module   return
caller's      function   not      while
```

"`caller's`" is a special case because it contains a non-alphanumeric apostrophe, yet it is treated as a single reserved word. No whitespace may appear around the apostrophe.

Some examples of identifiers:

Legal	Illegal
`alpha`	`1st`
`AlPhA`	`a_train`
`beta123z`	`say fromage`
`caller's`	

Here, `alpha` and `AlPhA` are the same identifier because of case insensitivity; `beta123z` is a legal identifier, as is the reserved word `caller's`. On the other hand, `1st` is erroneous since an identifier must start with an alphabetic character, `a_train` is illegal because of the underscore, and `say fromage` would be treated as two identifiers since whitespace may not appear inside an identifier.

String and Character Literals

String literals are denoted by quotation marks (“`"`”) on either side of zero or more character elements.

Character literals are denoted by single quotes (“`'`”) on either side of a single character element.

A string or character literal missing its terminating quote is an error. The contents of string and character literals *are* case sensitive, and unprintable characters, while discouraged, are permitted.

A character element may be either a single character or an escape sequence, consisting of a backslash followed by a single character:

`\a`	alert, typically ASCII BEL
`\b`	backspace
`\n`	newline
`\t`	horizontal tab
`\z`	zero char, i.e. ASCII NUL
`\"`	quotation mark
`\'`	single quote
`\\`	backslash

Any other escaped character is simply itself, without the backslash.

For example:

Legal	Illegal
`"Hello, world!\n"`	`"Goodbye...`
`"\"Duck!\""`	`'quack'`
`""`	`''`
`'a'`	
`'\\'`	

`"Goodbye...` is illegal because it is not terminated by a closing quote; both illegal character literals have the wrong number of character elements. Notice that an empty string literal is permitted, but an empty character literal is not.

Simple Lexical Elements

Besides the above, the following are also lexical elements:

```
=   ,   /   :    <=
;   +   %   :=   <>
(   -   .   ..   >
)   *   |   <    >=
```

Comments

Strictly speaking, comments are not separate lexical elements, but are a type of whitespace. A comment may be placed anywhere whitespace may be used, except in string or character literals. Comments begin with a hash mark (“`#`”), and cause the hash mark and all characters up to, but not including, the end of the line to be ignored.

Comments that begin with a hash mark immediately followed by "`output`" (case insensitive) and at least one whitespace character cause the remainder of the line, less leading whitespace, to be output *at compile time*.

This procedure illustrates the different comments:

```
procedure foo
    # This comment won't be output.

    #output This will be output at compile time.

    output("This will be output at run-time.\n");
end;
```

Miscellaneous

Reasonable length limitations may be imposed on lexical elements.

- C's `isspace()` macro from `ctype.h` is used to identify whitespace.
- Warnings are issued when newlines or unprintable characters are encountered in a string or character literal.
- No limitations are placed on comment length or consecutive whitespaces, but other elements are restricted to 1024 characters in length. Note that while string literals are restricted in length, instances of the predefined string type are unbounded.

Data Type Overview

This section provides a brief introduction to the data types available in MBL. A more precise syntactic and semantic definition is deferred to the *Annotated Grammar* section.

Integer

The maximum and minimum integer values should correspond to the target machine's usual word size. Overflow checking must be performed on integer values and all arithmetic operations involving them.

Character

A character must be large enough to hold all possible values in the target machine's character set. As with integers, overflow checking must be performed for all operations involving characters.

String

Strings may hold arbitrary sequences of character elements. There should be no length restrictions imposed on them. Operations on strings involving indices must be checked to ensure that a string's current length is not exceeded.

Note that strings are a basic type and not simply arrays of the character type.

Array

An array is a complex, fixed-sized data type. All elements of an array must be the same type, which may itself be arbitrarily complex. The number of elements in an array is fixed when the array is declared, and all array indices must be checked to ensure that they fall within the array bounds.

The lower bound on an array is always zero; the largest upper bound that may be declared is implementation-defined.

Note that while arrays may be only one-dimensional, a multi-dimensional array may be represented as an "`array of array of ...`".

Associative Array

Associative arrays are complex data types which map keys – strings – to values. Elements of an associative array must be of the same type. Associative arrays are indexed by strings, and their size is unbounded.

Much like a hash table, data stored in an associative array is not guaranteed to be in any particular order. The indexing of associative arrays should be a relatively inexpensive operation.

List

A list is a complex data type which behaves much like an unbounded array type. The elements of a list must be all of the same type.

Storage space for list elements should grow dynamically as required, and all list operations involving indices must ensure that a list's current length is not exceeded.

Record

A record groups data types into a larger unit, which is treated as a single structure. Each component of a record is called a ‘‘field’’, all of which must be given unique names to distinguish them from other fields. Individual fields in a record may be accessed by name.

- 32-bit integers are used.
- Eight-bit characters are used to hold all possible ASCII values.
- A maximum array size of 256 elements is allowed to be declared.

Typables

Many programming languages distinguish between data types – both predefined and user-defined – and variables. In those languages, a variable is an instance of a data type, therefore occupying storage space at run-time.

In MBL, structural type equivalence is used, meaning that two types are equivalent if and only if the structure of all their component parts is the same. Therefore, one object may be defined in terms of the structure of another: "`newtype: oldtype`" defines `newtype`'s structure to be the same as `oldtype`'s, so `newtype` has inherited the type from `oldtype`.

But if one object is defined in terms of another, then the only difference between a "data type" and a "variable" is that run-time storage space is allocated for the latter. MBL removes this distinction, combining types and variables into "typables".

A typable may be declared as having a particular type, and other typables may be defined in terms of it. However, *no run-time storage space is allocated for a typable unless it is used*. A typable is considered to be used if at least one of the following is true:

- the typable is referenced in a statement;
- the typable is the name of a formal parameter;
- the typable is exported;
- the typable is a for-loop variable.

For example,

```
module
     export foo: integer;
     import bar: char;

     baz: string:= "abc";

     procedure blarg(garble: integer)
          greep: baz;
          thisLittlePiggy: greep;

          greep:= "def";
     end;
end;
```

At run-time, `foo` is assigned storage space since it is exported, and other modules may be referencing it. `bar` also has storage space, but it is assigned by another module: the module `bar` is exported from. Even though `baz` has an expression to initialize it, it does not have storage space as it is neither exported nor used in a statement in the module.

As a formal parameter, `garble` has storage space (the exact storage location will depend on the calling conventions of the target machine). Finally, `greep` is given storage space because it is used in a statement, but `thisLittlePiggy` has none.

When `greep` is defined in terms of `baz`, the *only* thing that `greep` inherits is `baz`' type, *not* the initializer `"abc"` nor any attributes (like constant or import/export status).

Predefined Identifiers

Eight identifiers in MBL are "predefined", meaning their definitions exist prior to the first line of a MBL program being seen by the compiler:

```
atoi   halt    integer   output
char   input   itoa      string
```

For the most part, predefined identifiers associate names with data types and functions underlying the language, which would otherwise be inaccessible. *Note that these are not the same as reserved words.* Unlike reserved words, predefined identifiers may be redefined by the user of the language. Redefining a predefined identifier, however, causes any special association that that identifier had to be lost:

```
module
    oldinteger: integer;
    integer: string;

    procedure strange()
        old: oldinteger;
        new: integer;

        old:= 123;
        new:= 123;          # error
    end;
end;
```

In this rather confusing example, `oldinteger` is defined as `integer`'s type, namely the built-in integer type. Afterward, `integer` is redefined, but this doesn't affect `oldinteger`'s type nor the built-in integer type. This is demonstrated in the `strange` procedure, where a number can be assigned to `old`, but not `new`. So the identifier "`integer`" lost its special association with the built-in integer type through redefinition.

The predefined identifiers are defined as:

`integer`, `string`, and `char`

These predefined identifiers correspond to the built-in integer, string, and character data types. The *names* "`integer`", "`string`", and "`char`" may be redefined, but this does not affect the underlying built-in data types.

`halt`

This procedure outputs a supplied string as an error message and terminates execution of the program.

A declaration for `halt`, or any other predefined identifier, is not required – in fact, it would constitute a redefinition. However, for reference purposes, a declaration for `halt` would look like:

```
procedure halt(error: string);
```

`input` and `output`

These provide rudimentary I/O facilities. `input` is a function that returns a single input character as a string of length 1; the empty string is returned in case of error or end of file. `output` is a procedure which prints a string to the usual output device; a newline is *not* automatically generated after the string is printed.

Declarations for these two would look like:

```
function input(): string;
procedure output(s: string);
```

`atoi` and `itoa`

These are simple utility functions which convert between strings and integers.

`atoi` takes a string as its argument and returns an integer. The string may begin with an optional "+" or "−" sign, which must be followed by one or more digits. It is an error if the number represented by the string exceeds the maximum or minimum integer values.

`itoa`, on the other hand, takes an integer as an argument and returns a string. Their declarations would look like:

```
function atoi(s: string): integer;
function itoa(n: integer): string;
```

- Error messages from `halt` are output to `stderr`.
- `input` reads from `stdin` and `output` writes to `stdout`.

Scope

The scope of an identifier defines when it is visible – that is, when it can be referenced and used by other parts of the program. MBL uses static scope, meaning that an identifier's visibility is solely determined by its position in the program text.

A scope level is a section of program text which may contain:

- definitions of identifiers and those identifiers' complete scope;
- other scope levels.

An identifier's definition is a part of the scope level that was current when the identifier was first encountered in the program. The starting and ending points of a scope level correspond to five types of constructs: modules, procedures, functions, records, and for-loops. More precisely:

Element	Scope Level Begins	Scope Level Ends
module	after "`module`"	after module's "`end`"
procedure	before procedure's parameters	after procedure's "`end`"
function	before function's parameters	after function's "`end`"
record	after "`record`"	after record's "`end`"
for loops	after loop condition	after for loop's "`end`"

When an identifier is referenced in the program text, its definition is looked for in the current scope level, in the enclosing scope level if not found, and so on until either a definition is found or there are no more scope levels to search. Running out of scope levels in this manner is an error, because it means an undeclared identifier has been used. At run time, if there is more than one instance of the appropriate scope level (as can happen with recursion), then the most recent one should be referenced.

The scope of an identifier can now be understood in terms of scope levels. An identifier is visible from its definition point to the end of the scope level it is defined in. The definition points of various types of identifiers are below (note that the optional module identifier is not considered to be defined).

Identifier Type	Definition Point
typable	after complete type is known
field	after complete type is known
formal parameter	after complete type is known
procedure	after all formal parameters
function	after complete return type is known
for loop	after complete loop condition

Observe that, for procedures and functions, the beginning of the new scope level actually precedes the definition point of the procedure or function in the original scope.

With the exception of forward declarations for procedures and functions, identifiers may only be defined once per scope level.

Predefined identifiers are defined in a scope level which exists prior to the start of a module. The scope level begun by "`module`" is referred to as the "global" scope – as will be seen in the *Annotated Grammar* section, certain types of declarations may only be made in the global scope.

As a final clarification, MBL procedure and function declarations may be nested, up to a reasonable implementation-specific limit.

Annotated Grammar

Throughout this section, grammar symbols corresponding to lexical elements are placed entirely in uppercase, and may be treated as terminal symbols. In addition, identifiers (IDs) have been broken down based on the type of object the identifier represents:

Grammar Symbol	Lexical Element
NUMBER	number
STRING	string literal
CHARACTER	character literal
ID	identifier, must not be declared in current scope level
FUNCTIONID	identifier, most recently defined as a function
PROCEDUREID	identifier, most recently defined as a procedure
TYPABLEID	identifier, most recently defined as a typable
FIELDID	identifier, most recently defined as a field

Here, ‘‘most recently defined’’ simply refers to the identifier’s current definition with respect to the scope rules for looking up identifiers. (These rules were presented in the *Scope* section.)

The suffix ‘‘valued’’ is applied to the built-in data types (integer, string, and character) to describe something that is structurally equivalent to one of those types. So a ‘‘string-valued expression’’ is an expression whose type is structurally equivalent to the built-in string type.

For quick reference, the grammar rules appear without comments in the *Collected Grammar* section.

program $\rightarrow$ **‘`module`’ [ID] declarations ‘`end`’ ‘`;`’**

Input to a MBL compiler must contain exactly one module. One or more compiled modules (not necessarily all written in the same language) may be linked together by an external linker to form a complete executable program.

If the optional identifier is specified in the module declaration, it names a procedure or function where program execution is to begin. There must be exactly one starting point per executable.

The identifier must be the name of a procedure or function which is global and exported. It is an error if the named procedure or function is not defined in the module.

The target machine’s operating environment determines whether a procedure or function is required. A procedure is required for operating environments that do not expect an exit status from executables; a function is required for operating environments that need an exit status – the exact type of return value is implementation-specific. Regardless of operating environment, the procedure or function may not have any parameters.

declarations $\rightarrow$ **{ declaration }**
declaration $\rightarrow$ **[customs] declarationtype ‘`;`’**
customs $\rightarrow$ **‘`import`’ | ‘`export`’**

Declarations of global typables, procedures, and functions may be preceded by a customs declaration. A customs declaration may flag a typable, procedure, or function as one of:

Imported
: The object is defined in another module and may be used, but not defined, in the current module.

Exported
: The object must be defined in the current module and is usable by other modules. While any number of modules in an executable may import an object, only one may define and export it.

Objects not declared as exported or imported are not visible outside the current module. It is an error if an import or export attempt is made outside the global scope level.

Due to the above restrictions, importing and exporting are mutually exclusive. In the case of forward declarations, where an identifier may be defined more than once, any import or export attributes must agree between the two definitions.

declarationtype $\rightarrow$ **ID '`:`' ['`constant`'] typeinitialize**

This rule permits the declaration of typables – these identifiers must be previously undeclared at the current scope level. Those identifiers declared as structurally equivalent to the built-in data types (integer, character, and string) may be made constant – constant typables may not have their value changed or jeopardized (by passing them by reference, for instance). A constant must be initialized and may not be imported or exported.

declarationtype $\rightarrow$ **'`procedure`' ID '`(`' [formals] '`)`' [body]**
| **'`procedure`' PROCEDUREID '`(`' [formals] '`)`' body**
| **'`function`' ID '`(`' [formals] '`)`' '`:`' type [body]**
| **'`function`' FUNCTIONID '`(`' [formals] '`)`' '`:`' type body**
body $\rightarrow$ **declarations statements '`end`'**

Procedures and functions may be forward-declared, allowing a procedure or function to be used before being fully defined – this permits mutually recursive procedures and functions. A forward declaration is obtained simply by omitting the body from a regular procedure or function definition. A forward declaration must precede the actual declaration, and only one such declaration may the given for a particular procedure or function. The actual declaration must be at the same scope level as the forward declaration, and the two must match *exactly*, which means:

- any import or export status must be identical
- parameter names must be the same
- parameter types must be structurally equivalent
- by-reference status of all parameters must be identical
- function return types must be structurally equivalent

In general, except for forward declarations, you may only define an identifier once in a given scope level.

A function may be declared as returning any type – even complex ones like lists and arrays. Objects are always returned from functions by value. It is an error if a function doesn't execute a return statement.

Procedures and functions may be nested and called recursively. This example demonstrates the use of forward declarations and nested functions:

```
module
     function fibonacci(n: integer): integer;

     procedure tellfib()
          output(atoi(fibonacci(123)) + "\n");
     end;

     function fibonacci(n: integer): integer
          function fib(n: integer): integer
               if (n > 2)
                    return(n + fib(n - 1));
               end;
               return(1);
          end;

          return(fib(n));
     end;
end;
```

formals → **formal { '`,`' formal }**
formal → **ID '`:`' ['`caller's`'] type**

Formal parameters of procedures and functions may be of any type. Actual parameters are passed by value (even complex types) unless the reserved word "`caller's`" is used – this causes a parameter to be passed by reference.

Analogous to defining one typable in terms of another, declaring a parameter in terms of a typable only causes the parameter to be of the typable's type, and doesn't transfer any other attributes to the parameter. For example:

```
module
     x: constant integer:= 123;

     procedure foo(y: x)
          y := 456;
     end;
end;
```

Here, `y` only receives `x`'s type, not `x`'s constant status or its value, so `y` is an integer parameter.

typeinitialize → **type [initialize]**

initialize → **‘:=’ constantexpression**
| ‘:=’ constantexpression { ‘,’ constantexpression } ‘end’

Typables may be initialized as long as they do not contain associative arrays or lists, either directly or hidden in a component type. Imported typables may not be initialized; constant typables must be initialized.

An initialized typable has its value re-initialized upon every entry to the scope level the typable is declared in. This means that global typables are initialized prior to the program's execution; typables declared inside a procedure or function are initialized upon every invocation of that procedure or function. An uninitialized typable has an undefined initial value.

The number, order, and types of initializing expressions must exactly match those of the declared type:

```
x: array(2) of record
    c: char;
    i: integer;
end :=
    'a', 1,
    'b', 2
end;
```

The above code fragment shows `x` being initialized in this manner:

Element	Value
`x(0).c`	`'a'`
`x(0).i`	`1`
`x(1).c`	`'b'`
`x(1).i`	`2`

type → **‘array’ ‘(’ constantexpression ‘)’ ‘of’ type**

The constant expression in the array declaration defines the size of the array. This expression must be integer-valued, positive, and non-zero (as previously mentioned, the largest possible array size is implementation-defined).

The lower bound for arrays is always zero, meaning the valid array indices are from zero to the array size less one, inclusive.

type → **‘associative’ ‘array’ ‘of’ type**

Associative arrays are (theoretically) unbounded, and are indexed by string-valued expressions. Data stored in them is not guaranteed to be in any particular order, yet quick access to an associative array's elements should be ensured.

type → **‘record’ fields ‘end’**
fields → **field { field }**
field → **ID ‘:’ type ‘;’**

As discussed in the *Scope* section, records are like procedures, functions, and `for` loops in that their contents exist in a different scope level than the record is declared in. This record declaration is therefore legal:

```
blarg: record
    blarg: integer;
end;
```

Recursive record definitions are not permitted, since a record is not defined until the end of the record declaration is reached. The following, then, is illegal:

```
blarg: record
    garble: blarg;
end;
```

type $\rightarrow$ **'`list`' '`of`' type**

Lists are dynamic data structures which may be thought of as unbounded arrays. They are accessed with integer-valued expressions.

type $\rightarrow$ **TYPABLEID**

A type may be defined in terms of a previously-defined typable's type. Note that this is *only* a type assignment and that no attributes or initializing expressions are inherited. (Recall from the *Typables* section that a typable's attributes consist of constant and import/export status.)

constantexpression $\rightarrow$ **expression**

This grammar rule clarifies where constant expressions are expected.

An expression is a constant expression if it is comprised entirely of operations on constant typables and/or constant literal values. Taking the length of a (non-associative) array may be a component of a constant expression.

statements $\rightarrow$ **{ statement }**
statement $\rightarrow$ **statementtype '`;`'**

No statements are required, but empty statements are not allowed.

statementtype $\rightarrow$ **expression '`:=`' expression**

Assignment is permitted if the types of the expressions are structurally equivalent, and the left-hand-side expression is one of:

- a typable which is not constant; or
- a typable which is not constant, with one or more field or array references applied (see the *reference* grammar rule)

Assignment must always behave as though a copy of the right-hand-side's value is made, no matter how complex the type being assigned.

statementtype $\rightarrow$ **'`if`' '`(`' expression '`)`' statements ['`else`' statements] '`end`'**

Integer-valued MBL expressions are considered false if zero, and true if non-zero. A true expression causes the if-part statements to be executed; a false expression causes the optional else-part statements to be executed. The expression must be integer-valued.

statementtype → **‘`while`’ ‘`(`’ expression ‘`)`’ statements ‘`end`’**

Until the integer-valued expression is false, the statements will be repeated. If the expression is false initially, the statements will not be executed.

statementtype → **‘`case`’ ‘`(`’ expression ‘`)`’ cases [‘`else`’ statements] ‘`end`’**
cases → **case { case }**
case → **constantexpression ‘`:`’ statements ‘`end`’ ‘`;`’**

If the case expression matches any of the constant case label expressions, then the statements associated with the case label will be executed. If no matches are made, then the optional else-part statements are executed. It is an error if no match is found and no else-part is present.

The types of the case labels must be structurally equivalent to that of the case expression, and no two case labels may have the same value. The case expression may only be integer-, string-, or character-valued.

statementtype → **‘`for`’ ‘`(`’ ID ‘`in`’ loopcondition ‘`)`’ statements ‘`end`’**
loopcondition → **expression**
| **expression ‘`..`’ expression**

The for loop identifier exists in a different scope, as discussed in the *Scope* section. The identifier’s type is implicitly defined by the type of the loop condition. A loop condition may be either:

expression
The expression’s type must be an array, list, or associative array. The loop identifier will have the same type as the expression’s component type, and will assume a single value from the array, list, or associative array each iteration.

Expression Type	Loop Variable Type
list of X	X
array of X	X
associative array of X	X

This loop condition behaves as though a copy is made of each array, list, or associative array value, so the loop identifier may be altered inside the for loop without changing the original value. Arrays and lists are traversed in order from smallest to largest index; associative arrays have no such ordering guarantee.

The effect of modifying a list or associative array while it is being used in a loop condition’s expression is undefined. If there are no elements in the loop condition (e.g. an empty list), then the loop statements are not executed.

expression `..` expression
The two expressions must be both integer-valued or both character-valued. The loop identifier will have the same type as the expressions.

The loop identifier is initialized to the value of the leftmost expression before any of the loop statements are executed. The loop behaviour depends on the initial values of the leftmost and rightmost expressions:

leftmost ≤ rightmost
: The loop identifier is incremented at the end of an iteration.

leftmost > rightmost
: The loop identifier is decremented at the end of an iteration.

The loop terminates when the value of the loop identifier exceeds the value of the rightmost expression, which is re-evaluated every iteration.

The effect of modifying the loop identifier with this loop condition is undefined.

statementtype → PROCEDUREID '`(`' [expressions] '`)`'

This statement invokes a previously-declared procedure. A procedure must have the same number of actual parameters as formal parameters, and their corresponding types must be structurally equivalent. Actual parameters which are constant expressions may not be passed by reference.

statementtype → '`return`' '`(`' [expression] '`)`'

Usage depends on context. If used in a procedure, no expression is allowed; in a function, the type of the expression must be structurally equivalent to the function's declared return type.

statementtype → '`break`' '`(`' [constantexpression] '`)`'

This statement "breaks" out of enclosing while, for, or case statements, causing execution to resume at the first statement following the while, for, or case.

An omitted constant expression implies the value 1; a supplied constant expression must be integer-valued, positive, and non-zero. The constant expression specifies the number of enclosing while, for, or case statements to break out of – it is an error if at least the number specified do not exist.

expressions → expression { '`,`' expression }
expression → orexpression

An expression is really an "or-expression", but is separated for clarity in the grammar. An expression may have any valid type (see the *type* grammar rule). It is important to note that *type restrictions on an operator are only in effect if that operator is used*. If an operator is not used, any subexpression type passes through it unscathed.

Operator precedence, as defined in this grammar, is summarized below.

`| |`	– (unary)	`()`	`.`	highest precedence
`*`	`/`	`%`		
`+`	`–`			

```
= <> <= >= < >
not
and
or                          lowest precedence
```

orexpression → andexpression { ‘`or`’ andexpression }
andexpression → notexpression { ‘`and`’ notexpression }
notexpression → [‘`not`’] cmpexpression

These operators perform boolean and, or, and not. If applied, they require integer-valued operands, and always return an integer: false (zero) or true (non-zero). Boolean operators are short-circuiting.

“Short-circuiting” means that the minimum amount of evaluation is performed in order to obtain a result. Consider the expression “`a or b and c`”: if `a` is true, nothing else need be evaluated; if `a` and `b` are false, then `c` and the boolean “and” operator need not be evaluated. `c` is only relevant to the expression result if `a` is false and `b` is true.

cmpexpression → addexpression [comparison addexpression]
comparison → ‘=’ | ‘<>’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’

If a comparison operator is used, both operands must be of the same type. The nature of the comparison depends on the type involved:

= and <> (not equal)

integer and character
: A simple comparison; character comparisons are case-sensitive.

string
: A case-sensitive comparison.

anything else
: Complex types are equal if all their component parts are equal.

<=, >=, <, and >

integer and character
: A simple comparison; once again, character comparisons are case-sensitive.

string
: A case-sensitive lexicographic comparison.

anything else
: An error.

Like boolean expressions, comparisons always return false (zero) or true (non-zero) integer values. Whitespace is significant in string comparisons.

addexpression → mulexpression { addition mulexpression }
addition → ‘+’ | ‘−’

If applied, addition and subtraction operators may only be used in the combinations listed below.

Left Operand	Op	Right Operand	Result Type	Operation	Notes
integer	+	integer	integer	addition	1
integer	+	character	character	addition	1
character	+	integer	character	addition	1
character	+	character	character	addition	1
character	+	string	string	prepending	2
string	+	character	string	appending	2
string	+	string	string	concatenation	2
X	+	list of X	list of X	prepending	2
list of X	+	X	list of X	appending	2
list of X	+	list of X	list of X	concatenation	2
integer	-	integer	integer	subtraction	1
integer	-	character	character	subtraction	1
character	-	integer	character	subtraction	1
character	-	character	character	subtraction	1
integer	-	string	string	substring	2, 3
string	-	integer	string	substring	2, 3
integer	-	list of X	list of X	sublist	2, 4
list of X	-	integer	list of X	sublist	2, 4

Note 1
: Result values must not exceed the maximum or minimum values of the result type.

Note 2
: Result values must behave as copies – the operands cannot appear to have been modified by the operation.

Note 3
: The substring operation returns the leftmost or rightmost portions of a supplied string. For example:

```
"foo" - 2 = "f"
2 - "foo" = "o"
"foo" - 3 = ""
2 - "blarg" - 1 = "ar"
```

The integer value may not exceed the length of the string.

Note 4
: The sublist operation returns the leftmost or rightmost elements of a supplied list. It behaves in the same manner as the substring operation above, and is subject to the same error.

mulexpression → **factor { multiplication factor }**
multiplication → **'`*`' | '`/`' | '`%`'**

If applied, these operators perform multiplication, division, and remainder operations on integer-valued expressions. Their result type is always integer, and results may not exceed the maximum or minimum integer values. Division or remainder operations with a divisor of zero are an error.

factor → **‘|’ expression ‘|’**

The length operator cannot be applied to records, but it has a meaningful integer result for other operand types:

Type	Operation
integer	absolute value
character	ordinal (ASCII) value
string	string length
array	declared array size
list	list length
associative array	number of array elements

factor → **‘−’ factor**

Unary minus only works on integer-valued operands and returns an integer result. The result value must not exceed the maximum or minimum integer value.

factor → **NUMBER**

A factor may be a number literal, which is converted into an integer as required. A number literal, possibly combined with a unary minus operator, must be able to represent the *entire* range of integer values, but may not exceed the minimum or maximum integer bounds.

factor → **CHARACTER**
| STRING
| refexpression reference

A factor may be a character or string literal, or an expression qualified by zero or more field or array references.

refexpression → **‘(’ expression ‘)’**
| TYPABLEID

These rules permit nested expressions and typables to be expressions.

refexpression → **FUNCTIONID ‘(’ [expressions] ‘)’**

The semantics for a procedure’s actual parameters also apply to function invocations. The result type is the declared type of the function.

reference → **{ ‘.’ FIELDID | ‘(’ expression ‘)’ }**

A field reference may only be applied to a record type, and the field name must match a declared field name in the record. The result type is the type of the field.

An array reference may be applied to four types:

Type	Result Type	Notes
string	character	1

list of X	X	1
array of X	X	1
associative array of X	X	2

Note 1

The indexing expression must be integer-valued, and the index may not be less than zero nor greater than or equal to the length of the array, string, or list.

Note 2

The indexing expression for associative arrays must be string-valued. The element being referenced must exist unless the array reference is the last (or only) component of the expression on the left-hand-side of an assignment statement.

- A UNIX implementation requires a global, exported function returning an integer to be named by the optional module identifier.
- The reference implementation catches the complete absence of a return statement from a function at compile time, and a function not executing a return statement is caught at run-time.

Collected Grammar

program → ‘module’ [ID] declarations ‘end’ ‘;’
declarations → { declaration }
declaration → [customs] declarationtype ‘;’
customs → ‘import’ | ‘export’
declarationtype → ID ‘:’ [‘constant’] typeinitialize
| ‘procedure’ ID ‘(’ [formals] ‘)’ [body]
| ‘procedure’ PROCEDUREID ‘(’ [formals] ‘)’ body
| ‘function’ ID ‘(’ [formals] ‘)’ ‘:’ type [body]
| ‘function’ FUNCTIONID ‘(’ [formals] ‘)’ ‘:’ type body
body → declarations statements ‘end’
formals → formal { ‘,’ formal }
formal → ID ‘:’ [‘caller’s’] type
typeinitialize → type [initialize]
initialize → ‘:=’ constantexpression
| ‘:=’ constantexpression { ‘,’ constantexpression } ‘end’
type → ‘array’ ‘(’ constantexpression ‘)’ ‘of’ type
| ‘associative’ ‘array’ ‘of’ type
| ‘record’ fields ‘end’
| ‘list’ ‘of’ type
| TYPABLEID
fields → field { field }
field → ID ‘:’ type ‘;’
constantexpression → expression
statements → { statement }
statement → statementtype ‘;’
statementtype → expression ‘:=’ expression
| ‘if’ ‘(’ expression ‘)’ statements [‘else’ statements] ‘end’
| ‘while’ ‘(’ expression ‘)’ statements ‘end’
| ‘case’ ‘(’ expression ‘)’ cases [‘else’ statements] ‘end’
| ‘for’ ‘(’ ID ‘in’ loopcondition ‘)’ statements ‘end’
| PROCEDUREID ‘(’ [expressions] ‘)’
| ‘return’ ‘(’ [expression] ‘)’
| ‘break’ ‘(’ [constantexpression] ‘)’
loopcondition → expression
| expression ‘..’ expression
cases → case { case }
case → constantexpression ‘:’ statements ‘end’ ‘;’
expressions → expression { ‘,’ expression }
expression → orexpression
orexpression → andexpression { ‘or’ andexpression }
andexpression → notexpression { ‘and’ notexpression }
notexpression → [‘not’] cmpexpression
cmpexpression → addexpression [comparison addexpression]
comparison → ‘=’ | ‘<>’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’
addexpression → mulexpression { addition mulexpression }
addition → ‘+’ | ‘−’
mulexpression → factor { multiplication factor }

multiplication → ‘*’ | ‘/’ | ‘%’

factor → ‘|’ expression ‘|’

| ‘−’ factor

| NUMBER

| CHARACTER

| STRING

| refexpression reference

refexpression → ‘(’ expression ‘)’

| TYPABLEID

| FUNCTIONID ‘(’ [expressions] ‘)’

reference → { ‘.’ FIELDID | ‘(’ expression ‘)’ }

Examples

This section contains a number of MBL programs which are intended to give the reader a feel for the language. All these programs were written using the UNIX implementation of MBL, so the routine named by the module identifier is a globally-defined, exported function returning an integer – this detail would vary between operating environments.

These examples were graciously supplied by Shannon Jaeger; any sources she used are noted in header comments. This first program computes mean and variance of a series of space-separated numbers which is terminated by a period:

```
# Statistical calulations based on those in:
#    Introduction to Probability and Statistics 7th ed.
#    W. Mendenhall
#    PWS Publisher, 1983

module doStats
     inputList: list of integer;

     procedure readInput()
          newValue: string;

          string:= input();
          while (string <> ".")
               newValue:= newValue - (|newValue| - 1);
               while ((string <> " ") or (string <> "."))
                    if (string(0) < '0') or (string(0) > '9')
                         halt("error: non-numeric input");
                    end;
                    newValue:= newValue + string(0);
               end;
               inputList:= inputList + atoi(newValue);
          end;
     end;

     function calMean(): integer
          sum: integer;

          for (entry in inputList)
               sum:= sum + entry;
          end;
          return(sum / (|inputList| - 1));
     end;

     function calVariance(mean: integer): integer
          sum: integer;

          for (entry in inputList)
               sum:= sum +
                    (((entry - mean) * (entry - mean)) /
                    (|inputList| - 1));
```

```
        end;
        return(sum);
    end;

    export function doStats(): integer
        mean: integer;
        variance: integer;

        readInput();
        mean:= calMean();
        variance:= calVariance(mean);
        output("The Mean is: " + itoa(mean) + "\n");
        output("The Variance is: " + itoa(variance) + "\n");
        return(0);
    end;
end;
```

This sorts a list of integers using Quicksort:

```
# Based on algorithm in Introduction to Algorithms by T.H. Cormen,
# C.E. Leiserson and R.L. Rivest.  Published by The MIT Press,
# Cambridge, Massachusetts, 1991.

module
    aList: list of integer;

    function partition(pivot: integer, region: integer): integer
        pivotElement: integer;
        upperPar1: integer;
        lowerPar2: integer;
        tmp: integer;

        pivotElement:= aList(pivot);
        upperPar1:= pivot - 1;
        lowerPar2:= region + 1;
        while (1)
            lowerPar2:= lowerPar2 - 1;
            while (aList(lowerPar2) <= pivotElement)
                lowerPar2:= lowerPar2 - 1;
            end;
            upperPar1:= upperPar1 + 1;
            while (aList(upperPar1) >= pivotElement)
                upperPar1:= upperPar1 + 1;
            end;
            if (upperPar1 < lowerPar2)
                tmp:= aList(upperPar1);
                aList(upperPar1):= aList(lowerPar2);
                aList(lowerPar2):= tmp;
            else
                return(lowerPar2);
            end;
        end;
```

```
    end;

    procedure sortIt(pivot: integer, region: integer)
        newPivot: integer;

        if (pivot < region)
            newPivot:= partition(pivot, region);
            sortIt(pivot, newPivot);
            sortIt(newPivot+1, region);
        end;
    end;

    export
    function quicksort(theList: list of integer): list of integer
        aList:= theList;
        sortIt(0, |aList| - 1);
        return(aList);
    end;
end;
```

This program demonstrates MBL's scoping rules:

```
# This is based on the scoping examples, p. 158, in
# Pascal: An Introduction to the Art and Science of Programming,
# 2nd edition, by Walter J. Savitch.  Published by The
# Benjamin/Cummings Publishing Company, California, 1987.

module showScope
    x: integer;

    procedure ProA()
        x: integer;
    end;

    procedure ProB()
        x: integer;
    end;

    export function showScope(): integer
        procedure ProB()
            x: integer;

            procedure ProA()
                x:=integer;

                output("Start Procedure A\n");
                x := 3;
                output("In Procedure A, X = " +
                    itoa(x) +
                    "\n");
                output("End of Procedure A\n");
            end;
```

```
            output("Start procedure B\n");
            x:= 2;
            output("In Procedure B, X = " +
                itoa(x) +
                "\n");
            ProA();
            output("In Procedure B, X = " +
                itoa(x) +
                "\n");
            output("End of Procedure B\n");
        end;

        output("Start Program\n");
        x:= 1;
        output("Global x = " + itoa(x) + "\n");
        ProB();
        output("Global X = " + itoa(x) + "\n");
        output("End of Program\n");
        return(0);
    end;
end;
```

Note the use of associative arrays and by-reference parameters in this next example, which computes the change to issue for a given amount:

```
# This is based on example on page 127 of
# Pascal: An Introduction to the Art and Science of Programming,
# 2nd edition, by Walter J. Savitch.  Published by The
# Benjamin/Cummings Publishing Company, California, 1987.

module ComputeCoin
    coin: associative array of integer;
    amount: integer;

    procedure getInput()
        new: string;

        output("Enter an amount from 1 to 99 cents: ");
        string:= input();
        while (string <> "")
            new:= new + string;
            string:= input();
        end;
        amount:= atoi(string);
        if ((amount < 1) or (amount > 99))
            output("invalid input\n");
            getInput();
        end;
    end;

    procedure outputResult()
        output(itoa(amount) + " cents can be given as:\n");
```

```
        output("\t" + itoa(coin("quarters")) + "quarters\n");
        output("\t" + itoa(coin("dimes")) + "dimes\n");
        output("\t" + itoa(coin("nickels")) + "nickels\n");
        output("\t" + itoa(coin("pennies")) + "pennies\n");
    end;

    procedure computeQuarters(aLeft: caller's integer)
        coin("quarters"):= aLeft / 25;
        aLeft:= aLeft % 25;
    end;

    procedure computeDimes(aLeft: caller's integer)
        coin("dimes"):= aLeft / 10;
        aLeft:= aLeft % 10;
    end;

    procedure computeNickels(aLeft: caller's integer)
        coin("nickels"):= aLeft / 5;
        aLeft:= aLeft % 5;
    end;

    procedure computePennies(aLeft: caller's integer)
        coin("pennies"):= aLeft;
        aLeft:= 0;
    end;

    export function ComputeCoin(): integer
        aLeft: integer;

        coin("quarters"):= 0;
        coin("dimes"):= 0;
        coin("nickels"):= 0;
        coin("pennies"):= 0;

        getInput();
        aLeft:= amount;
        computeQuarters(aLeft);
        computeDimes(aLeft);
        computeNickels(aLeft);
        computePennies(aLeft);
        outputResult();
        return(0);
    end;
end;
```

This last example uses lists of records to maintain students' quiz scores:

```
# Based on program on page 412 of
# Pascal: An Introduction to the Art and Science of Programming,
# 2nd edition, by Walter J. Savitch.  Published by The
# Benjamin/Cummings Publishing Company, California, 1987.
```

```
module quizScores
     numStudents: constant integer:= 4;
     numQuizzes: constant integer:= 3;
     minScore: constant integer:= 1;
     maxScore: constant integer:= 10;
     student: record
          quiz: array(numQuizzes) of integer;
          mean: integer;
     end;
     gradeBook: array(numStudents) of student;

     procedure readQuizzes()
          inputString: string;

          for (counter1 in 0..numStudents-1)
               output("Enter " + itoa(numQuizzes) + " quiz scores");
               output("for student number " + itoa(counter1) + "\n");
               for (counter2 in 0..numQuizzes-1)
                    string:= input();
                    while ((string <> " ") and (string <> ""))
                         if ((string < "0") or (string > "9"))
                              halt("error: illegal input");
                         end;
                         inputString:= inputString + string;
                         string:= input();
                    end;
                    if ((inputString < itoa(minScore))
                         or (inputString > itoa(maxScore)))

                         halt("error: invalid score");
                    end;
                    gradeBook(counter1).quiz(counter2):=
                         atoi(inputString);
               end;
          end;
     end;

     function quizAve(who: student): integer
          sum: integer:= 0;

          for (quizMark in who.quiz)
               sum:= sum + quizMark;
          end;
          return(sum / numQuizzes);
     end;

     function classAve(class: array(numStudents) of student): integer
          sum: integer:= 0;

          for (who in 0..numStudents-1)
```

```
            for (quizMark in class(who).quiz)
                sum:= sum + quizMark;
            end;
        end;
        return(sum / (numStudents * numQuizzes));
    end;

    export function quizScores(): integer
        classAverage: integer;
        quizMarks: string;

        readQuizzes();
        for (person in 0..numStudents-1)
            gradeBook(person).mean:= quizAve(gradeBook(person));
        end;
        classAverage:= classAve(gradeBook);

        output("Student\tAve\tQuizzes\n");
        for (i in 0..numStudents-1)
            for (j in 0..numQuizzes-1)
                quizMarks:= quizMarks +
                    itoa(gradeBook(i).quiz(j)) +
                    " ";
            end;
            output(itoa(i) +
                "\t" +
                itoa(gradeBook(i).mean) +
                "\t" +
                quizMarks +
                "\n");
        end;
        output("The class average is: " + itoa(classAverage) + "\n");
        return(0);
    end;
end;
```

References

[1] N. Wirth, ‘‘What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?,’’ *CACM*, vol. 20, no. 11, pp. 822-823, November 1977.

[2] G. Kane and J. Heinrich, *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[3] MIPS Computer Systems Inc., *MIPS Assembly Language Programmer's Guide*, Sunnyvale, CA, 1991.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, 2nd Edition,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[5] S. C. Johnson, ‘‘YACC: Yet Another Compiler-Compiler,’’ *Unix Programmer's Manual*, vol. 2B, Bell Laboratories, Murray Hill, NJ, 1975.

[6] J. R. Levine, T. Mason, and D. Brown, *lex and yacc, 2nd Edition,* O'Reilly and Associates, Sebastopol, CA, 1992.