The Compiler Language: Syntax and Semantics

Everything is case in-sensitive: while is the same as WHILE; cat, Cat, and CAT are all the same thing. It is also free-form: any sequence of spaces, newlines, and comments is considered to be exactly the same as a single space

Statements

As usual, statements may only appear inside function bodies. A semicolon is not part of a statement, it is a separator for statements, so is not required before a \}. For convenience, it is also not required immediately after a \} either.

<table>
<thead>
<tr>
<th>IF statement</th>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
</table>
| IF expression1 THEN statement1 ELSE statement2 | Code = S_IF  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for statement1  
Part[2] = ptr to node for statement2 |

| IF expression1 THEN statement1 | Code = S_IF  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for statement1 |

Rules are as normal, except for one improvement. To prevent ambiguity (the dangling else problem), statement1 may not be another IF statement. This is not a restriction: statement1 may be an IF inside { }, which makes it a block.

<table>
<thead>
<tr>
<th>WHILE statement</th>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
</table>
| WHILE expression1 DO statement1 | Code = S_WHILE  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for statement1 |

Rules are as normal. The expression is tested every time before the statement is executed. If the expression is initially false, the statement is not executed at all.
The variable must already be declared, this loop does not introduce a new local variable.

Before the loop starts, the variable is given the value of expression1.

Each time around the loop, before executing statement1, expression2 is re-evaluated, and variable is compared with it.

In the case of TO, the loop terminates if variable is greater than expression2.

In the case of DOWNTO, the loop terminates if variable is less than expression2.

Each time around the loop, immediately after executing statement1, the variable is updated. In the case of TO, it is incremented by 1; in the case of DOWNTO, it is decremented by 1.

When the loop is finished, the controlling variable retains its last value. If the loop terminated because of a BREAK or a RETURN, it keeps the value it had at that time. If the loop terminated because of comparison with expression2, the variable has the value it had when that comparison was made – that is, the value that made the loop stop.

Modifying the controlling variable inside the loop has no surprising effects.

---

**BREAK statement**

Syntax | syntax_element node representation.
---|---
BREAK | Code = S_BREAK
Rules are as normal. The BREAK statement is only allowed inside a loop which is itself inside the current function. It causes the immediate termination of the smallest enclosing loop, having no effect on any outer loops. In the case of a FOR loop, the controlling variable is unaffected by a BREAK.

**CONTINUE statement**

Syntax | syntax_element node representation.
---|---
CONTINUE | Code = S_CONTINUE

Rules are as normal. The CONTINUE statement is only allowed inside a loop which is itself inside the current function. The current iteration of that loop is terminated, but the loop itself is continued.

In the case of a WHILE loop, the test is performed again immediately, and the loop ends if the condition is false.

In the case of a FOR loop, the controlling variable is incremented or decremented in the normal way, and the comparison with the end value performed again. The loop terminates if the end value has been passed.

**EXIT statement**

Syntax | syntax_element node representation.
---|---
EXIT | Code = S_EXIT

The program is immediately terminated, regardless of any loops or functions that may be active.

**RETURN statement**

Syntax | syntax_element node representation.
---|---
RETURN | Code = S_RETURN

RETURN expression1 | Code = S_RETURN
Part[0] = ptr to node for expression1
If there is an expression, it is evaluated, the enclosing function call is terminated, and the value of the expression is used as the value of the function.
If there is no expression, the enclosing function call is terminated, and its value is indeterminate.

### OUTPUT statements

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTN expression1</td>
<td>Code = S_OUT</td>
</tr>
<tr>
<td></td>
<td>Info = ‘N’</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for expression1</td>
</tr>
<tr>
<td>OUTCH expression1</td>
<td>Code = S_OUT</td>
</tr>
<tr>
<td></td>
<td>Info = ‘C’</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for expression1</td>
</tr>
<tr>
<td>OUTS expression1</td>
<td>Code = S_OUT</td>
</tr>
<tr>
<td></td>
<td>Info = ‘S’</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for expression1</td>
</tr>
</tbody>
</table>

The expression is evaluated, and its value is displayed on the controlling terminal in one of these three way:
OUTN The value is printed as a decimal integer
OUTCH One character is printed, the one whose ASCII code is given by the value
OUTS The value is assumed to be the address of a string somewhere in memory. Characters starting from that position are printed as with OUTCH, until a zero code is reached. The zero code is not printed.

### ASSIGNMENT statement

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvalue1 = expression1</td>
<td>Code = S_ASSIGN</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for lvalue1</td>
</tr>
<tr>
<td></td>
<td>Part[1] = ptr to node for expression1</td>
</tr>
</tbody>
</table>


An lvalue is an expression that represents a location in memory, such as a variable name or an array access. The expression is evaluated and stored as a single word in the memory location denoted by the lvalue. An assignment will not copy an array or a string, just a single word value.

**FUNCTION CALL statement**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
</table>
| expression1() | Code = S_FUNCTION_CALL  
Part[0] = ptr to node for expression1 |
| expression1(expression2) | Code = S_FUNCTION_CALL  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for expression2 |
| expression1(e2, e3) | Code = S_FUNCTION_CALL  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for e2  
Part[1] = ptr to node for e3 |
| expression1(e2, e3 e4, ..., en) | Code = S_FUNCTION_CALL  
Part[0] = ptr to node for expression1  
Part[1] = ptr to node for e2  
Part[2] = ptr to node for e3  
Part[3] = ptr to node for e4  
.....  
Part[n-1] = ptr to node for en |

Expression1 provides the function to be called. Usually it will simply be the name of a function, but it may be a variable that contains the address of a function, or any other expression whose value is the address of a function.

The parameters are passed using value semantics. That is, they are all evaluated exactly once, and their values are used to initialise local variables inside the function itself.

If the function returns a value, it is ignored.
BLOCK statement

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Code = S_BLOCK</td>
</tr>
<tr>
<td>{ statement1 }</td>
<td>Code = S_BLOCK</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for statement1</td>
</tr>
<tr>
<td>{ statement1 ; statement2 }</td>
<td>Code = S_BLOCK</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for statement1</td>
</tr>
<tr>
<td></td>
<td>Part[1] = ptr to node for statement2</td>
</tr>
<tr>
<td>{ statement1 ; statement2 ; statement3 ; ..... statementn }</td>
<td>Code = S_BLOCK</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for statement1</td>
</tr>
<tr>
<td></td>
<td>Part[1] = ptr to node for statement2</td>
</tr>
<tr>
<td></td>
<td>Part[2] = ptr to node for statement3</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>Part[n-1] = ptr to node for statementn</td>
</tr>
</tbody>
</table>

The statements are executed in order. Anything declared within a block is local and temporary, there is no effect on similarly named variables already in existence, and no residual effect after the end of the block. All exactly as usual,

LOCAL declaration

See the description of variable declarations.

In-line ASSEMBLY code

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[ assemblycode ]]</td>
<td>Code = S_ASSEMBLY</td>
</tr>
<tr>
<td></td>
<td>Info = index number in vector</td>
</tr>
</tbody>
</table>

The assemblycode may be absolutely anything. The compiler does not process it except as follows:
Any appearance of [], even in a comment, marks the end.
Any text within [ and ] is replaced as shown below.
The content is pasted by the compiler directly in the output (assembly) file. Any
errors must be detected by the assembler when the whole file is assembled.

Inside the assembly code, [ and ] may surround a single simple name. If that name
refers to a local variable or a parameter in the current context, the whole is replaced
by the correct frame-pointer-relative address (e.g. $FP – 2). If the name is that of a
function or global variable, it is replaced by the correct internal name. It is for the
programmer to remember to use * as necessary.

Declarations

Declarations define local and global variables and arrays, and functions.
Functions may only be declared at the top level, i.e. not inside any other function.
The standard scope rules apply.

Local declarations are performed when they are executed.

Global declarations are performed simultaneously. This means that functions can
be defined in any order without the need for prototypes. The declare-before-use
rule does not apply to globals. Functions may refer to global variables that have
not yet been declared. The initialising values of globals, if any are provided, must
be static constants.

CONSTANT declarations

Syntax

<table>
<thead>
<tr>
<th>Code</th>
<th>Syntax Element Node Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL item1</td>
<td>S_VAR_DECL</td>
</tr>
<tr>
<td>LOCAL item1, item2</td>
<td>S_VAR_DECL</td>
</tr>
</tbody>
</table>

Declarations define local and global variables and arrays, and functions.
Functions may only be declared at the top level, i.e. not inside any other function.
The standard scope rules apply.

Local declarations are performed when they are executed.

Global declarations are performed simultaneously. This means that functions can
be defined in any order without the need for prototypes. The declare-before-use
rule does not apply to globals. Functions may refer to global variables that have
not yet been declared. The initialising values of globals, if any are provided, must
be static constants.

CONSTANT declarations

Syntax

<table>
<thead>
<tr>
<th>Code</th>
<th>Syntax Element Node Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL item1</td>
<td>S_VAR_DECL</td>
</tr>
<tr>
<td>LOCAL item1, item2</td>
<td>S_VAR_DECL</td>
</tr>
</tbody>
</table>

Const declarations, as expected, define named constants. They may be declare
both globally and locally. The initial value must be a static constant, that is a
number, another named constant, or an expression consisting only of such things.

Named constants occupy no memory in a running program, thus they do not
have addresses. They may be used as array sizes in declarations.

VARIABLE declarations

Syntax

<table>
<thead>
<tr>
<th>Code</th>
<th>Syntax Element Node Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL item1</td>
<td>S_VAR_DECL</td>
</tr>
<tr>
<td>LOCAL item1, item2</td>
<td>S_VAR_DECL</td>
</tr>
</tbody>
</table>
Local and global declarations have exactly the same form. The word LOCAL may only be used inside a function. The word GLOBAL may only be used at the top level, i.e. not in any function.

There are three possible forms for the individual declaration items:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Code = S_VAR_DECL_ITEM</td>
</tr>
<tr>
<td></td>
<td>Sym = ptr to symbol descr. for name</td>
</tr>
<tr>
<td></td>
<td>Info = 0</td>
</tr>
<tr>
<td>name = expression1</td>
<td>Code = S_VAR_DECL_ITEM</td>
</tr>
<tr>
<td></td>
<td>Sym = ptr to symbol descr. for name</td>
</tr>
<tr>
<td></td>
<td>Info = 0</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node for expression1</td>
</tr>
<tr>
<td>name [ number ]</td>
<td>Code = S_VAR_DECL_ITEM</td>
</tr>
<tr>
<td></td>
<td>Sym = ptr to symbol descr. for name</td>
</tr>
<tr>
<td></td>
<td>Info = the number</td>
</tr>
</tbody>
</table>

The first form, just a name, introduces a new variable. It occupies one word and is not initialised.

The second form, name=value, introduces a new variable. It occupies one word and is initialised to the given value. In the case of a local variable, it is initialised
every time its enclosing block is entered. In the case of a global variable, it is only initialised once, when the program is loaded.

The third form, name[N], introduces a new array. It is of size N (valid indices run from 0 to N-1), but it occupies N+1 words of memory. The name refers directly to the first word, as though it were a simple variable. This first word contains the address of the next word, which is where the array proper begins. The result is that array variables may be passed to functions and used normally, and they may be assigned to, to make them refer to different arrays. The size N must be a static constant.

The second and third forms may not be combined: arrays can not be given initial values, they must be assigned values in the normal way.

<table>
<thead>
<tr>
<th>FUNCTION declarations</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION name(a, b, ..., c) { statements }</td>
<td>Code = S_FUNC_DEF</td>
</tr>
<tr>
<td></td>
<td>Sym = ptr to symbol descr. for name</td>
</tr>
<tr>
<td></td>
<td>Part[0] = ptr to node parameter names</td>
</tr>
<tr>
<td></td>
<td>Part[1] = ptr to node for the block</td>
</tr>
</tbody>
</table>

Function definitions may only be at the global scope; there are no prototypes, nor is there any need for them.

The list of parameters is represented internally by an S_PARAM_NAMES node. The list may be empty, just ( followed by ), but even then it is represented by an S_PARAM_NAMES node.

As there are no types, nothing can be checked when a function is called. A function may safely be called with any number of parameters, regardless of how many appeared in the function’s declaration, but care must be taken not to attempt to access parameters that were not passed. There is no way to tell how many parameters were actually passed.

A function that returns a value is no different from one that doesn’t.

A parameter-names node has Code=S_PARAM_NAMES. The only other information is in its Part vector. For the n\textsuperscript{th} parameter name, Part[n-1] is a pointer to a syntax_element representing the name itself:

- Code = S_IDENTIFIER
- Sym = pointer to symbol_description for the name
MAIN declarations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>syntax_element node representation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code = S_FUNC_DEF</td>
<td></td>
</tr>
<tr>
<td>MAIN</td>
<td>Sym = ptr to symbol description for “main”</td>
</tr>
<tr>
<td>{ statements }</td>
<td>Part[0] = ptr to empty param names node</td>
</tr>
<tr>
<td></td>
<td>Part[1] = ptr to node for the block</td>
</tr>
</tbody>
</table>

MAIN is very much like a function, but its declaration is made to stand out. The word FUNCTION is not used, MAIN is a reserved word. MAIN has no parameters, it will not even accept an empty ( ). MAIN may be defined anywhere in the program (at global scope, of course), top, bottom or in the middle.

Arrays

Arrays are created as in these examples:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOCAL list[length]</td>
</tr>
</tbody>
</table>

In each of these example cases, an array large enough to hold 5 words is created. The correct indexes are from 0 to 4. Possible accesses include:

```
FOR i=0 TO 4 DO
  arr[i] = i*i
  OUTN list[x]
```

Although the arrays have space for 5 words, they occupy 6 words in memory. The first of those 6 always contains the address of (or a pointer to) the rest. This means that the name of an array refers to the whole array, not one element of it. For example, the third example happened to occupy memory locations 361 to 366, the arrangement in memory would be:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>accessed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>366</td>
<td>uninitialised</td>
<td>list[4]</td>
</tr>
<tr>
<td>365</td>
<td>uninitialised</td>
<td>list[3]</td>
</tr>
<tr>
<td>364</td>
<td>uninitialised</td>
<td>list[2]</td>
</tr>
<tr>
<td>363</td>
<td>uninitialised</td>
<td>list[1]</td>
</tr>
<tr>
<td>362</td>
<td>uninitialised</td>
<td>list[0]</td>
</tr>
<tr>
<td>361</td>
<td>362</td>
<td>list</td>
</tr>
</tbody>
</table>

When an array access such as A[x] is performed, the values of A and X are simply added together to find the address of the appropriate array item.

The significance of this is that storing and array variable in another variable, such as with the assignment B=list, makes the new variable behave like the whole array. B[x] will be exactly the same thing as list[x]. That means that arrays may be passed
as parameters to functions without any need for pointer operations or type declarations.

Structures and selectors

Structures or objects should be implemented as ordinary arrays of the right size to hold everything, together with pre-defined selectors for the fields. These selectors would normally be named constants.

As an example, a struct that contains a pointer to a string (1 word), two sixteen bit values, and a normal (1 word) number, would occupy 3 words, and may be set up like this:

```
CONST object_size = 3;
CONST object_str = WORD 0;
CONST object_x = BITS 32 TO 47;
CONST object_y = BITS 48 to 63;
CONST object_num = WORD 2;
```

There are three ways of creating a selector:

- BIT n is equivalent to BITS n TO n
- WORD n is equivalent to BITS 32*n TO 32*n+31

The result of a BITS, BIT, or WORD expression is an ordinary number, in which the most significant 8 bits and the least significant 24 bits play separate parts. The first contains the length in bits of the item being described; the second contains the number of bits between the beginning of the object and the beginning of this item.

```
BITS 6 TO 13  =  0x08000006    (length is 8, start bit is 6)
```

Items inside objects may be accessed using the FROM operator. “A FROM B” may be used both as a value and as a destination for assignment. In either case, A must be a well-formed selector value, and B must be the address of an object (which is most likely to be an array variable). Continuing the example, an object may be created and used like this:

```
LOCAL item[object_size];
object_x FROM item = 123;
OUTS  object_str FROM item;
BITS 64 TO 67 FROM item = 5;
```

The FROM operator expects its right operand to be the address of a (potentially large) area of memory. The OF operator works in the same way, but expects its right operand to be a simple value.

```
BITS 16 TO 27 FROM 0x926B4A7F
```

would extract its 12 bits from the contents of memory location 0x926B4A7F.

```
BITS 16 TO 27 OF 0x926B4A7F
```
would extract its 12 bits from the value 0x926B4A7F, yielding 0x4A7. The OF operator may also be used in a destination. BIT 31 OF x = 0; would ensure that the value of variable x is even (it sets the least significant bit to zero).

For convenience, the dot and arrow operators are also provided:

A . B  is equivalent to  B OF A
A -> B  is equivalent to  B FROM A

Expressions in order of priority

Atomic expressions:
name  variable name, function name, array name, etc
numeric constant  0, 12345, -7, etc
               numbers may begin with  0x or 0h for base 16,
               0b for base 2,
               0o for base 8.
               in all other cases, a leading zero is meaningless.
string constant  "cat", "", "aefj vw3ukl fjr", etc
( expression )  another expression in parentheses.
true  equivalent to 1
false  equivalent to 0
null  equivalent to 0

Priority 1:
expression ( )
expression ( expression )
expression ( expression , expression ) and so on: function calls
expression [ expression ]  array access
expression . expression  see structs description, above
expression -> expression  see structs description, above

Priority 2:
BITS expression TO expression
BIT expression
WORD expression

Priority 3:
expression OF expression
expression FROM expression
Priority 4:
- * expression
  - follow pointer
- & lvalue
  - find address / make pointer to
- @ lvalue
  - identical to &

Priority 5:
- - expression
- + expression

Priority 6:
- expression * expression
- expression / expression
- expression % expression

Priority 7:
- expression + expression
- expression - expression

Priority 8:
- expression = expression
- expression == expression
  - equivalent to =
- expression < expression
- expression <= expression
- expression > expression
- expression >= expression
- expression != expression
  - equivalent to !=
- expression <> expression
  - equivalent to !=

Priority 9:
- NOT expression
- ! expression
  - equivalent to NOT

Priority 10:
- expression AND expression
  - equivalent to AND
- expression && expression
  - equivalent to AND
- expression & expression
  - equivalent to AND

Priority 11:
- expression OR expression
  - equivalent to OR
- expression | | expression
  - equivalent to OR
- expression | expression
  - equivalent to OR