**Introduction to MORSE LANG**

Morse lang is a esoteric, high-level programming language that is based on Morse code. Every single use of Morse code in this language is sourced from International Morse Code.

This language introduces several odd features not found in most programming languages. Examples of this includes: variables may have spaces and all keywords and variables are separated by slashes (“/”) to mimic how Morse code is written.

The following is an overview of what can be found so far in the language (Up to Compiler level 4)

* multi-line **comments**
* literals for scalar **datatypes** integer, boolean, string.
* Explicitly defined scalar **variables** with datatypes: integer and boolean.
* **Global** scoped and function **local** scoped variables
* A list of unary and binary operators
* INPUT-statement and PRINT-statement for console formatted text input/output.
* **Multi-function** programs
* A **DO-WHILE** statement
* **IF, IF ELSE,** and **ELSE** statements
* **FOR** statements
* **Assertions**
* **Multi-module** call-by-value programs
* **CALL** and **RETURN** statements for subprograms
* **Directly-recursive** program modules

**Semnatics**

Because this language takes inspiration from Morse code, letters are separated by spaces and “words” or tokens are separated by forward slashes.

**Comments**

A multi-line (block) **<comment>** may appear between any tokens. The BNF below:

**<comment> ::= << { <ASCIICharacter> }\* >>**

may be used as

-... . --. .. -./<<prints 'Hello World'>>/..-. ..- -. -.-./Main/.--. .-. .. -. -/\*Hello World\*/--..--/-. .-../-.-/.-.-.-

**Identifiers**

An **<identifier>** is used to name variables, functions, formal parameters; and subprogram module variables. A subprogram module identifier is defined when it is specified in the header of the module’s definition. A variable is defined when it is specified in a data definitions section. A formal parameter is defined when it is specified in a subprogram module formal parameter list.

Identifiers must start with an alphanumerical character and cannot contains forward slashes (/). Due to the way the language is parsed, identifiers may have spaces in their names.

The scope of global variables is global scope and extends from the point of definition to the end of the source program.

The scope of function variables and subprogram module formal parameters is local scope and extends from the point of definition to the end of the module body.

The **lifetime** (or **duration**) of global variables and constants and of program module local variables and constants is the entire duration of the program’s execution. The lifetime of subprogram module local variables (including formal parameters) and constants begins when the module is referenced (that is, when the subprogram module’s activation record is created on the run-time stack) and continues until the module returns flow-of-control to the module which referenced it (that is, until the subprogram module’s activation record is destroyed).

A scalar **<varaiable>** is **referenced** by specifying its identifier. An array **<variable>** element is referenced by specifying the array identifier suffixed with integer index expressions enclosed in square brackets [...]. It is a static semantic error when the number of integer index expressions does not match the number of dimensions specified when the array variable is defined. It is a **run-time error[[1]](#endnote-2)** when any of the integer index expression values are not in the range of index bounds specified when the array variable is defined. An array can only be referenced as a whole entity (that is, without the suffixed comma-separated list of integer index expression enclosed in square brackets) when it is being used as an actual parameter which corresponds to a REF formal parameter because arrays can only be passed-by-reference.

A scalar variable and an array variable element must be used as an **l-value** or an **r-value**; a scalar constant can only be used as an r-value (an l-value can be changed, an r-value cannot). Note A subprogram module formal parameter is treated as a locally-defined

**<variable> ::= <identifier> [[ <expression> ]]**

**Program**

A **<Program>** is an optional collection of 1 or more global data definitions; followed by an optional collection of 0 or more subprogram module definitions; followed by a required program module definition. When the program begins execution, flow-of-control begins with the first statement in the program module’s list of statements and flow-of-control continues until it terminates when the statement which ends the list of program module statements completes execution.

**<Program> ::= { <dataDefinition> } \***

**{ (( <FUNCTIONDefinition> )) }\***

**<MAINDefinition>**

**Function**

A **<FUNCTIONDefinition>** defines a named subprogram module by providing a globally-unique identifier, an optional list of formal parameters, and optional set of data definitions, and an optional set of executable statements within the subprogram module

**<FUNCTIONDefinition> ::= ..-. ..- -. -.-. / <identifier> / -.--. /**

**( <formalParameter> {, <formalParameter> }\* )**

**/ -.--.- /**

**{ <dataDefinition> }\***

**{ <statement> }\***

**Main**

A **<MAINDefinition>** defines a unique subprogram module with an optional set of data definitions and an optional set of executable statements within. This module is unique to the program but is always required, much like a classical main function.

**<MAINDefinition> ::= -- .- .. -. /**

**{ <dataDefinition> }\***

**{ <statement> }\***

**.-.-.-**

**Data Definitions**

A <dataDefinition> is a list (empty or not) of variables. Variables are named by an **<identifier>** , have an explicit type, and may have an initial value or not, and could be either scalar or an array.

All variables are defined in a **<dataDefinition>** block that precedes all statements in the program and have a global scope. The scope of variables defined in a **<functionDefinition>** have local scope, beginning at the start of the function and ending with the end of the function. A formal parameter is defined in the parameter list of a function.

**<dataDefinition> ::= <variableDefinition> | <constantDefinition>**

**<variableDefinition> ::= <datatype> <identifier> [ [ <LBUBRange> ] ]**

**<datatype> ::= ( .. -. - | -.. -... .-..**

**| ... - .-. | -... --- --- .-..**

**| -.-. .... .-.)**

**Functions**

A **<FUNCTIONDefinition>** defines a program module. Because tokens are not separated by spaces, identifiers can be written with a space. A number of formal parameters may be specified. The BNF below:

**<FUNCTIONDefinition> ::= ..-. ..- -. -.-. <identifier>**

**< ( <formalParameter> {, <formalParameter> }\* ) >**

**{ <dataDefinition> }\***

**{ <statement> }\***

may be used as:

-... . --. .. -./<<prints 'Hello World'>>/..-. ..- -. -.-./Custom Function Name/.--. .-. .. -. -/\*Inside Custom Function\*/--..--/-. .-../-.-/.-.-.-

**Expressions**

An **<expression>** computes a scalar ineger, double, or boolean. The BNF below:

**<expression> ::= <conjuction> { (( | | )) }\***

**<conjuction> ::= <negation> / { (( --- .-. | -. --- - ))**

**/ <negation> }\***

**<negation> ::= [ -. --- - ] <comparison>**

**<comparison> ::= <comparator> [ / (( )) / <comparator> ]**

**<comparator> ::= <term { (( + | - )) <term> }\***

**<term> ::= <factor> { (( –- | -.. )) <factor> }\***

**<factor> ::= [ .-.-. | -....- ] <variable>**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Precedence** | **Associativity** |
| “--” or “-..” | 1 | Left-to-right |
| “.-.-.” or “-….-” | 2 | Left-to-right |
| “.-.. -”, “.-.. - -...-”, “--. -”, “--. - -...-”, “-...- -...-”, or “-. -...-” | 3 | Left-to-right |
| “-. --- -” | 4 | Left-to-right |
| “.- -. -..” | 5 | Left-to-right |
| “--- .-.” | 6 | Left-to-right |

an expression may be expressed as:

-... . --. .. -./..-. ..- -. -.-./Main/.--. .-. .. -. -/-. .-../--..--/\*10/3 = \*/--..--/-.--./10/-../3/-.--.-/--..--/-. .-../-.-/.-.-.-

**Printing**

The **<PRINTStatement>** adds the computed value of integer, double, boolean expressions, and string litterals to the console output window. The BNF below:

**<PRINTStatment> ::= .--. .-. .. -. - / (( <string> | <expression> )) { --..-- / (( <string> | <expression> )) }\* / -.-**

can be expressed as:

-... . --. .. -./<<prints 'Hello World' with newline>>/..-. ..- -. -.-./Main

/.--. .-. .. -. -/\*Hello World\*/--..--/-. .-../-.-/.-.-.-

**Inputing**

The **<INPUTStatement>** takes a **<string>** and an **<identifier>**. It prompts the user with the string and will insert what the user enters into the identifier. The datatypes MUST match or else an error will be thrown.

**<INPUTStatment> ::= .. -. .--. ..- - / <string> / <variable> / -.-**

Below is an example of code:

-... . --. .. -./..-. ..- -. -.-./Input Example

/.. -. -/int1/-.-

/.. -. .--. ..- -/\* Input int1: \*/int1/-.-

/.--. .-. .. -. -/-. .-../--..--/\*int1 = \*/--..--/-.--./int1/-.--.-/--..--/-. .-../-.-

/.-.-.-

**Assignment**

The **<assignmentStatement>** is a fairly straightforward combination of a identifier and a value.

**<assignmentStatement> ::= <variable> / -...- / <expression> / -.-**

Below is an example of code:

-... . --. .. -./..-. ..- -. -.-./Morse Code

/.. -. -/int1/-...-/4/-.-

/.--. .-. .. -. -/-. .-../--..--/\*int1 = \*/--..--/-.--./int1/-.--.-/--..--/-. .-../-.-

/.-.-.-

**If Statement**

The **<IFStatement>** acts like a typical If-Statement. When the expression of the if-statement is true, the statements within the if-statement is evaluated. When the expression is false, the flow of control skips to the next Else or Else If Statement, if available.

**<IFStatment> ::= .. ..-. / -.--. / <expression> / -.--.- /**

**{ <statment> }\***

**{. .-.. ... . .. ..-.**

**/ -.--. / <expression> / -.--.-**

**/ { <statment> }\* }\***

**[. .-.. ... .**

**/ { <statment> }\* ]**

**/ .-.-.-**

**Do-While**

The **<DOWHILEStatement>** is an unbounded, post-test loop.

The statements in the Do-While loop are executed and the while expression is evaluated. If the expression is true, then the statements are re-executed, and when the expression is false, the flow-of-control continues past the do-while loop.

**For**

The **<FORStatement>** is equivalent to a classically bounded pre-test loop.

**<FORStatement> ::= ..-. --- .-. / < <expression> / -.-**

**/ <expression> / -.-**

**/ <expression> > /**

**| { <statement> }\* |**

**/ -.-**

The expressions are evaluated once before executing the body of the loop. The first **<expression>** is lower-bound when the third expression is positive but upper-bound when the third expression is negative. The second expression is uppper-bound when the third expression is positive but lower-bound when the third expression is negative. The first expression dictates the starting value of the bounded variable, the second expression dictates the upper or lower bound of the loop, and the third expression dictates by at what interval the variable is increased for every loop.

**Assertion**

The **<assertion>** statement contains an expression that is evaluated once an assertion is encountered. If the expression evaluates to true, then the flow-of-control continues on. If the expression is evaluated to false, then the program terminates.

**<assertion> ::= .- ... .-. - / -.--. /<expression> / -.--.-**

**Call**

The **<CALLStatement>** calls a function by it’s identifier. It can provide several parameters to pass into the function.

**<CALLStatement> ::= -.-. .- .-.. .-.. / <identifier> / -.-**

**Return**

The **<RETURNStatement>** returns the flow of control to where the function was initially called.

**<RETURNStatement> ::= .-. - .-. -. / [ ( <expression> ) ] /** **-.-**

**BNF**

**<Program> ::= { <dataDefinition> } \***

**{ (( <FUNCTIONDefinition> )) }\***

**<MAINDefinition>**

**<dataDefinition> ::= <variableDefinition>**

**<variableDefinition> ::= <datatype> <identifier> [ [ <LBUBRange> ] ]**

**<datatype> ::= ( .. -. - | -.. -... .-..**

**| ... - .-. | -... --- --- .-..**

**| -.-. .... .-.)**

**<FUNCTIONDefinition> ::= ..-. ..- -. -.-. / <identifier> / -.--. /**

**( <formalParameter> {, <formalParameter> }\* )**

**/ -.--.- /**

**{ <dataDefinition> }\***

**{ <statement> }\***

**.-.-.-**

**<MAINDefinition> ::= -- .- .. -. /**

**{ <dataDefinition> }\***

**{ <statement> }\***

**.-.-.-**

**<formalParameter> ::= [ <identifier> <datatype> [ | {,}\* | ] ]**

**<statment> ::= { <assertion> }\***

**(( <PRINTStatement> | <INPUTStatment>**

**| <assignmentStatement> | <IFStatement>**

**| <FORStatement> | <DOWHILEStatement>**

**| <CALLStatement> | <RETURNStatement>**

**)) { <assertion> }\***

**<assertion> ::= .- ... .-. - / -.--. /<expression> / -.--.-**

**<assignmentStatement> ::= <variable> / -...- / <expression> / -.-**

**<PRINTStatment> ::= .--. .-. .. -. - / (( <string> | <expression> )) { --..-- / (( <string> | <expression> )) }\* / -.-**

**<INPUTStatment> ::= .. -. .--. ..- - / <string> / <variable> / -.-**

**<IFStatment> ::= .. ..-. / -.--. / <expression> / -.--.- /**

**{ <statment> }\***

**{. .-.. ... . .. ..-.**

**/ -.--. / <expression> / -.--.-**

**/ { <statment> }\* }\***

**[. .-.. ... .**

**/ { <statment> }\* ]**

**/ .-.-.-**

**<DOWHILEStatement> ::= -.. --- /**

**| { <statement> }\* |**

**/ .-- .... .. .-.. . / < <expression> > / -.-**

**<FORStatement> ::= ..-. --- .-. / < <expression> / -.-**

**/ <expression> / -.-**

**/ <expression> > /**

**| { <statement> }\* |**

**/ -.-**

**<CALLStatement> ::= -.-. .- .-.. .-.. / <identifier> / -.-**

**<RETURNStatement> ::= .-. - .-. -. / [ ( <expression> ) ][ / -.--.**

**/ (<expression> | <variable>){ / --..--**

**/ ( <expression> | <variable> ) } ] / -.--.-**

**/** **-.-**

**<expression> ::= <conjuction> { (( | | )) }\***

**<conjuction> ::= <negation> / { (( --- .-. | -. --- - ))**

**/ <negation> }\***

**<negation> ::= [ -. --- - ] <comparison>**

**<comparison> ::= <comparator> [ / (( )) / <comparator> ]**

**<comparator> ::= <term { (( + | - )) <term> }\***

**<term> ::= <factor> { (( –- | -.. )) <factor> }\***

**<factor> ::= [ .-.-. | -....- ] <variable>**

**<identifier> ::= (( <letter> )) { (( <letter> | <digit> | \_ )) }\***

**<literal> ::= <integer> | <double> | <boolean> | <string>**

**<integer> ::= <digit> { <digit> }\***

**<float> ::= <integer> , <integer>**

**<variable> ::= <identifier> [[ <expression> ]]**

**<boolean> ::= - .-. ..- . | ..-. .- .-.. … .**

**<string> ::= \* / { <ASCIICharacter> }\* / \***

**<letter> ::= A | B | ... | Z | a | b | ... | z**

**<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

**<ASCIICharacter> ::= || Every printible ASCII character in range**

**[‘ ’,’~’]**

**<comment> ::= << { <ASCIICharacter> }\* >>**

**KEYWORDS**

**BEGIN (BEGIN): -... . –-. .. -.**

**INTEGER (INT): .. -. -**

**DOUBLE (DBL): -.. -... .-..**

**STRING (STR): ... - .-.**

**BOOLEAN (BOOL): -... --- --- .-..**

**CHARACTER (CHR): -.-. .... .-.**

**ASSIGNMENT (=): -...-**

**END STMT (K): -.-**

**FUNCTION (FUNC): ..-. ..- -. -.-.**

**MAIN FUNCTION (MAIN): -- .- .. -.**

**END FUNCTION (PERIOD): .-.-.-**

**COMMA: --..--**

**IF (IF):** **.. ..-.**

**ELSE IF (ELSEIF):** **. .-.. ... .**

**ELSE (ELSE): . .-.. ... .**

**DO (DO): -.. ---**

**WHILE (WHILE): .-- .... .. .-.. .**

**FOR (FOR): ..-. --- .-.**

**PRINT (PRINT): .--. .-. .. -. -**

**INPUT (INPUT): .. -. .--. ..- -**

**RETURN (RTRN): .-. - .-. -.**

**CALL (CALL): -.-. .- .-.. .-..**

**OPEN PARENS ((): -.--.**

**CLOSE PARENS ()): -.--.-**

**TRUE (TRUE): - .-. ..- .**

**FALSE (FALSE):** **..-. .- .-.. ... .**

**OR (OR): --- .-.**

**AND (AND): .- -. -..**

**NOT (NOT): -. --- -**

**MULTIPLY (M): --**

**DIVIDE (D): -..**

**ADD (+): .-.-.**

**MINUS (-): -....-**

**NEW LINE (NL): -. .-..**

**QUOTE (“): .-..-.**

**LESSTHAN (LT): .-.. -**

**LESSTHANEQUAL (LT=): .-.. - -...-**

**EQUAL (==): -...- -...-**

**GREATERTHAN (GT): --. -**

**GREATERTHANEQUAL (GT=): --. - -...-**

**NOTEQUAL (N=): -. -...-**

**ASSERT (ASRT): .- ... .-. -**

**References**

Source of Morse code information: <https://morsecode.scphillips.com/morse2.html>

1. [↑](#endnote-ref-2)