I. LANGUAGE OVERVIEW

Overview - Mission Briefing

Militari is a general-purpose, procedural, imperative computer programming language inspired by common military terms and commands and based from C Programming Language - the most widely used computer language. The language itself is patterned in the same way of creating a program in C Language

This document is designed for software programmers with a need to understand the Militari programming language starting from scratch. This will give you enough understanding on Militari programming language from where you can take yourself to higher level of expertise.

Before proceeding with this document, you should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages particularly C will help you in understanding the programming concepts and move fast on the learning track.

II. GENERAL RULES

15 General Orders

1. **Militari** is a case sensitive programming language. Thus, 'Variable' and 'variable' are two different identifiers.
2. All global declarations must be declared outside of **PrimaryMission** and any other sub functions. Declaration must be in this order: data type, identifier name equals value.
3. Whitespace is required between keywords and identifier.
4. Any space or special character, such as %, & within the identifier name will be considered an error. Additionally, reserved words must not be used as identifier.
5. The program must have a **PrimaryMission** function. The keyword **PrimaryMission** is followed by () and a newline space before the opening { symbol of its body and must end with its closing symbol }. The program will compile with no errors if the **PrimaryMission** function is empty.
6. Compiler ignore anything written as comment in your program.
7. Maximum length of naming an identifier is 31.
8. Every line of code must end with a semicolon.
9. Subfunctions must also be defined before the **PrimaryMission** function. A function can be called under a function.
10. To comment a word or a message simply enclose it in between of two **@** symbols.
11. Arithmetic Expressions cannot be used to initialize values for variables.
12. An array can only have a maximum of two dimensions only.
13. Negative values must be preceded with a tilde (~).

14. Conditional and looping statements must begin with open curly brace ("{") before putting simple and compound statements and end with ("}").
15. The whole program must be terminated by **deploy()** keyword.

III. STRUCTURE OVERVIEW

Programming Structure

Before we introduce the basic building blocks of Militari programming language, let us look a bare minimum Militari program structure so that we can take it as a reference in upcoming pages.

A Militari program basically consists of the following parts:

- Variable Declarations
- Subfunctions
- PrimaryMission Function
- End of program

```
<Constant Declaration>

<Global Declaration>
<Struct Declaration>

<Function Declaration>

PrimaryMission() {

<Local Declaration>

<Statements>

} deploy();
```

Let us look at a simple code that would print the words **“Hello World”**:

```
PrimaryMission() {
@Display Hello World@
post(“Hello World!”);
}
deploy();
```

Let us look at the various parts of the program:

1. The first line of the function PrimaryMission() is the main function where the program execution begins.
2. The next line @...@ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
3. The next line post(…) is another function available in Militari which causes the message “Hello World!” to be displayed on the screen.
4. The next line deploy(); terminates the program.

IV. BASIC SYNTAX

Programming Syntax

You have seen a basic structure of Militari program, so it will be easy to understand other basic building blocks of the Militari programming language.

TOKENS IN MILITARI

A Militari program consists of various tokens and a token is either a keywords, an identifier, a constant, a literal, or a symbol. For example, the following Militari statement consists of 5 tokens:

```
post( “ Hello World! ” );
```

The individual tokens are:

```
post
(
 “ Hello World ”
)
;
```

SEMICOLON

In a Militari program, the semicolon is a statement or line terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
For example, the following are two different statements:

```
post(“ Hello World! ”);
deploy();
```

COMMENTS

Comments are like helping text in your MILITIARI program and they are ignored by the compiler. They start with a @ and terminates with another @ as shown below:

```
@ My first Program in Militari @
```

IDENTIFIERS

An identifier are series of characters consisting of letters, digits, and underscores that is given to name a variable, constant, and function.

Rules:

· In declaring and using an identifier in the Militari language, it should start with any alphabet or underscore. The first letter can be followed by a lowercase or uppercase letter(s), a positive digit(s)/number(s), or an underscore. Any special characters such as $,#,& and/or @ will be considered errors.

· Identifier name must not be repeated. Therefore, unit var1; and digit var1; will produce an error.

Here are some examples of acceptable identifiers:

_variable	variable	v_ariable
VariableNum	variableNum	Variablenum

KEYWORDS

The following shows the reserved words in Militari. These reserved words may not be used as constant or variable or any other identifier names.

PrimaryMission	backup	unit	inorder	phase
post	campaign	company	otherorder	miss
capture	operation	digit	order	action
hold	abort	inquire	go	deploy
Extent	ToJoeRange	Carry	Swap	Commence

WHITESPACE IN MILITARI

A line containing only whitespace, possibly with a comment, is known as a blank line, and a Militari compiler totally ignores it. Whitespace is the term used in Militari to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement,

```
unit num1;
```

such as unit, ends and the next element begins. Therefore, in the following statement:

There must be at least one whitespace character (usually a space) between unit and num1 for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
sum = num1 + num2; @ Get the sum of num1 and num2 @
```

No whitespace characters are necessary between sum and =, or between = and num1, although you are free to include some if you wish for readability purpose.

INPUT AND OUTPUT STATEMENTS IN MILITARI

Capture statement is used for taking user input or reading data one by one.

Rules:

- **capture** function must be of lowercase letters. Any capitalization on the word c-a-p-t-u-r-e will produce an error.
- The keyword **capture** must only be followed by a space or an opening parenthesis “(”, followed by the identifier prefixed with ‘#’ that will act as placeholder for the input, and enclosed by a closing parenthesis “)”. The statement should always be terminated by semicolons “;”.
- Identifier, arrays and struct member are the only valid placeholders for the values to be input by the user. Other than that like, literals, will not be accepted.
- There should be only one collect statement for each user’s input and putting multiple placeholders inside the collect statement would produce an error.

SYNTAX:

```
capture( <placeholder> ) ;
```

EXAMPLE:

```
PrimaryMission(){
post(“Enter number”);
capture(#var1);
} deploy();
```

On the other hand, the *post* statement is used in displaying a series of characters or values of an identifier.

RULES:

- **post** function must be of lowercase letters.
- The keyword **post** must only be followed by an opening parenthesis "(", followed by the value to be printed out, and enclosed by a closing parenthesis ")". The statement should always be terminated by a semicolon ";".
- When you input stringlit (<stringlit>) you should use double quotation mark (" ")
  and for identifiers, you simply enclosed it in parenthesis
  ("("…")")
- In concatenating values inside the post statement, the ampersand symbol "+" should be used. Note that the "+" symbol must have a space in between separating the values to be concatenated.
- Newlines can be used in line with the string to be displayed. To display a new line, simply add \n inside the stringlit.

SYNTAX:

```
post( "Hello World!" )
```

V. DATA TYPES, VARIABLES

Programming Syntax

In the MILITARI programming language, data types refer to an extensive system used for declaring variables or functions of different types.

unit	defines variables holding whole numeric values	Integer
digit	has a floating value or a fractional part	Float
joe	stores a single character	Character
company	stores a sequence of characters	String
response	used for declaring true or false	Boolean

A variable is name given to a storage area that our programs can manipulate. Each variable in Militari has a specific type, which determines the size and layout of the variable; the range of values that can be stored within; and the set of operations that can be applied to the variable.

RULES IN NAMING A VARIABLE:

· Variables names must start with an underscore or any alphabet character followed by any lowercase or uppercase letter, a number or an underscore.

· Variable must have at least 2 minimum characters maximum of 31 characters.

RULES IN DECLARING A VARIABLE:

• Global declarations are only done before the PrimaryMission() function. Else, if a variable is declared either inside the PrimaryMission() function or subfunctions, it is considered locally declared.

- It is not allowed to declare variables with different data types in a single statement. Declaring it to the next line will do.
- A variable with data type UNIT and DIGIT has a default value of 0.
- A variable with data type of RESPONSE has a default value of NEGATIVE.
- The declaration of variable should be done before it can be used in a program.
- In declaring multiple variables, a comma is used to separate variable names from one another.
- It must be properly terminated using semicolon (“ ; “)
- Data types must be in lower case.

RULES IN INITIALIZING A VARIABLE:

- In Initializing a variable, there must be an equal sign between the variable and its value.
- In Initializing variables, there must be a comma separating from other variables and its respective value.
- In Initializing variables, take note that an arithmetic equation cannot be used.
- The value to be stored in the variable must be a valid literal for the data type of the identifier.

Some valid declarations are shown here:

```
unit var1 = 10;

digit _var2 = 10.5;

joe var_3 = ‘A’;

company vAr4 = “Hello World!”;

response vaR5 = AFFIRMATIVE;
```

1. UNIT

RULES:

- Declaring data type unit is case-sensitive and can only be written as “unit”.
- Only unit literals can be accepted as input.
- Uninitialized unit variable will be initialized zero by default

SYNTAX:

```
unit<identifier>;

unit <identifier>, <identifier>,…, <identifier>;

unit <identifier>=<numliteral> ;
```

EXAMPLE:

VALID	INVALID
unit var1 = 10;	unit var1 = 10.5;
unit var1, var2, var3;	uit var1, var2, var3;

2. DIGIT

RULES:

- Declaring data type digit is case-sensitive and can only be written as “digit”,
- Only digit literals can be accepted as input.

- Uninitialized digit variable will be initialized zero by default

SYNTAX:

```
digit<identifier>;

digit <identifier>, <identifier>,…, <identifier>;

digit <identifier>=<decliteral> ;
```

EXAMPLE:

VALID	INVALID
digit var1 = 10.5;	digit var1 = 10.5.5;
digit var1, var2, var3;	digt var1, var2, var3;

3. JOE

RULES:

- Declaring data type joe is case-sensitive and can only be written as “joe”,
- Only character literals can be accepted as input.
- Character must be enclosed in a pair of single quotes.
- Uninitialized joe variable will be initialized as a whitespace by default

SYNTAX:

```
joe <identifier>;

joe <identifier>, <identifier>,…, <identifier>;

joe <identifier>=<charliteral> ;
```

EXAMPLE:

VALID	INVALID
joe var1 = ‘A’ ;	joe var1 = “A” ;
joe var1, var2, var3;	jo var1, var2, var3;

4. COMPANY

RULES:

- Declaring data type company is case-sensitive and can only be written as “company”,
- Only string literals can be accepted as input.
- String must be enclosed in a pair of double quotes.
- Uninitialized company variable will be initialized as a whitespace by default

SYNTAX:

```
company <identifier>;

company <identifier>, <identifier>,…, <identifier>;

company <identifier>=<stringliteral> ;
```

EXAMPLE:

VALID	INVALID
company var1 = “Hello World” ;	company var1 = ‘Hello’ ;
company var1, var2, var3;	comany var1, var2, var3;

5. RESPONSE

RULES:

- Declaring data type response is case-sensitive and can only be written as “response”,
- Only boolean literals can be accepted as input.
- Uninitialized response variable will be initialized AFFIRMATIVE by default

SYNTAX:

```
response <identifier>;

response <identifier>, <identifier>,…, <identifier>;

response <identifier>=<booleanliteral> ;
```

EXAMPLE:

VALID	INVALID
response var1 = AFFIRMATIVE;	response var1 = TRUE;
response var1, var2, var3;	respnse var1, var2, var3;

VI. CONSTANTS, LITERALS

Programming Syntax

The constants refer to fixed values that the program may not alter during its execution. Constants can be of any of the basic data types like a unit constant, a digit constant, a joe constant, or a company literal. The constants are treated just like regular variables except that their values cannot be modified after their definition.

RULES:

- Value of constant declared identifier cannot be altered through the entire program.
- Constant declarations are made through "hold" statement. You have to initialize a value to a constant when it is declared.
- Declared only before the PrimaryMissions function in the program.
- Multiple initialization of constant declared value is not allowed.
- Constant declaration must match the data type and the value of the identifier.
- Identifiers to be declared as constant are declared globally.
- Each constant declaration consists of the reserved word hold. And it is followed by the data type, identifier, equal sign, value and the terminator semicolon (“;”).

SYNTAX:

```
hold <dataType> <identifier> = <value>;
```

EXAMPLE:

Valid	Invalid
hold unit xy = 23;	unit gem xy = 256;
hold digit xy = 23.5;	digit xy = 23.5;

UNIT LITERAL (Numlit)

numlit is a type of literal that signifies whole number.

RULES:

- The numlit is limited only up to 9 digits only ranging from ˜999,999,999-999,999,999.
- If the integral part of the numbers exceeded 9 places, it will result to lexical errors.
- numlit accepts only digits without decimal point.
- numlit accepts digits as positive digit/s or negative digit/s. Negative digits are indicated by using tilde (˜).By the absence of the negative symbol, the input digits will be read as positive.
- Negative symbol "˜" is only valid before a number in numlit.
- Zero is also accepted as numlit.
- Leading zeroes will not be accepted.
- Zeroes with negative sign will be considered as 0.

DIGIT LITERAL (Declit)

DIGIT literal is a type of literal that signifies decimal values.

RULES:

- Range of declit is from ˜99,999.9999-99,999.9999
- A declit is up to 4 decimal places.
- If the decimal part of the number exceed the four decimal places, it will be a lexical error.
- declit considers positive and negative digits. Negative digit/s are denoted by using tilde (˜) as a negative sign. Otherwise, it is a positive digit/s.
- Zero is accepted a declit.

- Only one leading zero before the decimal point is accepted as a declit.
- Succeeding zeroes and digits after the two decimal places will not be accepted and read.
- Zero with a negative sign (~) is considered an error.

JOE LITERAL

- Charlit is a type of literal that signifies a character in Militari language.

RULES:

- Charlit is any printable character except new line, tab and single quote.
- Alphabets and digits are allowed to be an joe literal.
- The length of a Charlit is one (1) only. Else, it will be considered as an error.
- a character will be considered as a Charlit if and only if it enclosed in a single quote.

COMPANY LITERAL (Stringlit)

Company literal is a type of literal that represents string values.

RULES:

- stringlit is any printable character except new line, tab and double quote.
- A character will be considered as a stringlit if and only if it enclosed in a double quote.

RESPONSE LITERAL

Response literal is a type of literal that signifies the logical values AFFIRMATIVE (true) or NEGATIVE (false).

RULES:

- A Response literal can only be AFFIRMATIVE or NEGATIVE.
- The default value of response should be AFFIRMATIVE (TRUE) in conditional statements and looping statements.

DATA TYPE	Valid	Invalid
unit	123456789	12345678.90
digit	1234.9090	12345.00023
joe	‘a’	‘aaa’ , aaa
response	AFFIRMATIVE,NEGATIVE	True,False
company	“aabbccee”	Asdsawds

VII. ARRAYS

Programming Syntax

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

RULES:

- The datatype and the variable name are separated by a space.
- The minimum size of array is (1) while the maximum limit for a single dimension is [100] and for two dimensional the limit will be [50] [50].
- Array indices are fixed numeric values as declared in its declaration.
- Array contents must strictly contain a data of its own datatype.
- Array indices are positive whole numbers only.
- When declaring unknown size, you should declare the set of elements.
- The maximum dimension of an array is two.

SYNTAX (ARRAY DECLARATION):

```
<datatype><identifier> [<arraysize>] ;
<datatype><identifier>[<arraysize>][<arraysize>] ;
```

SYNTAX (ARRAY INITIALIZATION):

```
<datatype><identifier> [<arraysize>] = {<elements>}
<datatype><identifier>[<arraysize>][<arraysize>] = { {<elements>,<elements>}
```

RULES:

- The values of array elements should match datatype of the declared array name.
- If the number of elements initialized is less than the size of the array, then the indexes that have no corresponding value will be: for unit and digit: 0, for joe and company: empty string and for response: is AFFIRMATIVE.
- Element of an array must be enclosed in a pair of curly braces (“{ “ , “}”) and each element is separated by comma (,).

SYNTAX (ARRAY ASSIGNMENT):

```
<identifier> [<array literal>] = {<value>};
```

EXAMPLE:

Valid	Invalid
joe astring [10];	joe astring xy = [256];
unit var[23] = { }	unit var[] = { 1,2,3,4,5}

.

VIII. OPERATORS

Programming Syntax

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

ARITHMETIC OPERATORS

These are symbols representing mathematical operation. The following table shows all the arithmetic operators supported by Militari language. Assume variable A holds 10 and variable B holds 5, then:

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides the second operand from the first operand	A / B will give 2
%	Modulus Operator and remainder of after an integer division	A % B will give 0
ˆ	Exponentiation, raises the base to its nth power	A ˆ B will give 100000
˜	Negation	˜A is equivalent to -10

RULES:

- The symbol "~" or tilde can be placed before a value; after the mathematical operator and must be followed by a value, variable or other mathematical expressions
- Expressions are solved according to the *PEMDAS* rule:
- The most inner expressions enclosed in numerous parentheses is the first expression that will be evaluated.
- Exponents – represents how many times a value should be multiplied. Base – either a number or a variable – should be placed before the "ˆ" operator and the power – can also be a number or a variable – on the other hand should be placed after the "ˆ" operator.
- An operator must be located between two values, expressions or may be followed by an open parenthesis.
- Numerous grouping op expressions is acceptable by the compiler. Group of expressions must be enclosed by the parenthesis.

UNARY OPERATORS

These are operators that act upon a single operand to produce a new value.

RULES:

- Unary operators are not allowed in a mathematical expression. It could only be used in an iteration on a looping statement.
- Unary operators are not allowed in a function

RELATIONAL OPERATORS

These are operators that pertains to the relation between two values or entities. Assume variable A holds 10 and variable B holds 5, then:

Operator	Description	Example
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true	A < B will give NEGATIVE
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	A > B will give AFFIRMATIVE
=>	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true	A => B will give AFFIRMATIVE
=<	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	A =< B will give NEGATIVE
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	A == B will give NEGATIVE
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	A != B will give AFFIRMATIVE

RULES:

- Comparing expressions including variables in compiler is allowed.
- Values of a variable should be same data type except for **unit** and **digit**.

- unit and digit can be compared using relational operators, while the company, joe, and response can only be compared using "==" and "!=".
- A relational operator must be located between two expressions.
- The use of grouping symbols, open and close parenthesis is allowed.

LOGICAL OPERATORS

These are operators that signifies the connection of two or more relational expressions. It produces win or lose values. Following table shows all the logical operators supported by Militari language. Assume variable A holds win and variable B holds lose, then:

Operator	Description	Example
&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	A & B will give NEGATIVE
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	A || B will give AFFIRMATIVE
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	A != B will give AFFIRMATIVE

ASSIGNMENT OPERATORS

There are following assignment operators supported by Militari language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	X = Y + Z will assign value of Y + Z into X

IX. CONDITIONS

Programming Syntax

These are features of programming language which perform different computations or actions depending on whether a condition evaluates to true or false.

RULES:

- Function can only accept relational and logical separators for evaluation.
- There must be always a Boolean expression enclosed in parentheses “()” after the word “inorder”, “otherorder”, “order”. Boolean expression can be either a Boolean value or a conditional expression. The arguments in the conditional expression could be an identifier, an array, a struct, a function or a literal, with a conditional/relational operator.
- Next to it, any number of statements must be enclosed in open and close braces symbol “{}”.
- Declaration inside the code block is optional.

EXAMPLE:

```
inorder( <condition> ){
<statements>
}
```

- Nested conditional statements are allowed in Militari language.

INORDER STATEMENT

An *inorder* statement consists of a boolean expression followed by one or more statements. A Boolean expression is can be either a boolean value or a conditional expression. If the condition next to the “inorder” keyword is true, the statements within the “inorder” block will be processed.

SYNTAX:

```
inorder( <condition> ){
<statements>
}
```

INORDER-ORDER STATEMENT

If the Boolean expression next to the “inorder” keyword is true, the statements within the “inorder” block will be processed. If not, statements under the “order” keyword will be processed.

SYNTAX:

```
inorder( <condition> ){
<statement/s>
}
order{
<statement/s>
}
```

INORDER-OTHERORDER-ORDER STATEMENT

If the condition next to the “inorder” keyword is true, the statements within the “inorder” block will be processed. If not, statements within the otherorder – which pertains to another condition

- will be executed. And if all conditions stated in “inorder” and “otherorder” did not met a condition, statement/s under the “order” keyword will be processed.

- The order statement can only be written after the inorder statement. Should there exist an otherorder statement, order statement will be written after it.
- Order statements must always be enclosed with ({…}). Order statement is executed when the condition in inorder and otherorder evaluates a value of NEGATIVE.

SYNTAX:

```
inorder ( <condition> ){
<statement/s>
}
otherorder ( <condition> ){
<statement/s>
}
order{
<statement/s>
}
```

SELECT-CASE STATEMENT

This statement provides multiple selections based on the value of the variable which passes the control to the selection where it matches.

RULES:

- The keyword “operation” must be followed by an identifier enclosed in parentheses “()”.

- For each operation keyword, it should be followed by a number or a character literal, then by a colon, followed by statements.
- The statements under the “action” keyword will be executed if none of the statements under “campaign” keywords above met the value of the identifier assessed in “operation” statement.
- No output will be produced if in case the “action” keyword is unavailable and/or none of the “campaign” options matched the value of identifier evaluated in the “operation” statement.
- The action statement must have at least a single statement. Else, it will produce an error.

- Case parameter in *operation()* statement are only positive integers and characters. ex. Case ‘A’ or Case 1 or Case ‘1’.
- Default in *operation()* statement is optional and can be used once inside the statement.
- Operation statement can be empty. Meanwhile, campaign statements must at least have one statement.
- The values that the identifier in the operation statement can be either a unit literal or joe literal.
- Keyword “abort” is used at the end of every statement in an operation.

SYNTAX:

```
campaign( <variable> ){
operation <Literal> :
      <statement/s>
      abort();
operation <Literal> :
      <statement/s>
      abort();
action:
      <statement/s>
```

X. LOOPS

Programming Syntax

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

A ‘loop’ is one structure in the code of a program that is executed repeatedly under a satisfied condition. Repetition is done once a condition is met or it becomes false.

RULES:

- Nested looping statements are allowed in Militari language.
- Declaration inside the code block is optional, but the block itself cannot be null.

PHASE STATEMENT

A control flow statement that permits code to be carry out repeatedly based on a given condition. A condition is first evaluated, and then if the code is true, the code within the block is executed. This loops until the condition becomes false.

RULES:

- Use the keyword “phase” before placing the condition that is enclosed in parentheses “()”.
- Statements must be enclosed in a pair of curly braces (“{“…”}”)
- While the condition evaluates to AFFIRMATIVE, the compiler will execute the phase statements and when the condition is evaluated as NEGATIVE, it will skip the statements included in the phase loop.

SYNTAX:

```
phase (<condition>){
<variable with initial value>
<statement>
<action>
}
```

SAMPLE:

```
phase(a!=5){
post(a);
a++;
}
```

GO PHASE STATEMENT

A control flow statement that permits code to be carry out repeatedly based on a given condition. The code within the re-search block is executed once, and then the condition is assessed. If the evaluated condition is true, the code within the block will be executed again until the condition becomes false.

RULES:

- The keyword “go” should be followed by condition enclosed in parenthesis.
- Statements must be enclosed in a pair of curly braces(“{“…”}”)
- While the condition evaluates to AFFIRMATIVE, the compiler will execute go Statements and while the condition is evaluated as NEGATIVE. It will execute now the phase loop.

SYNTAX:

```
go {
<variable with initial value>
<statement>
<action>
} phase (<condition>)
```

SAMPLE:

```
go {
post(a);
a++;
} phase (a!=5)
```

INQUIRE STATEMENT

Same with the other looping statements mentioned above, it is control flow statement that permits code to be carried out repeatedly based on a given condition. The code within the rotate block is executed one at a time until the condition becomes false.

RULES:

- There are 3 expressions consists in “inquire” loop, separated by a semi-colon “;” in the control block of this loop all enclosed in parentheses “()”.
- The first expression is the counter initialization which is executed first in the control statement. This is use for initializing the value of the variable for the series of loop.

- The second condition is the test condition. This will act as the token whether the loop’s execution should continue or not based on the result of the condition’s assessment – either it is true or false.
- The third expression is the update expression. This will tell whether the loop will decrease or increase in value.
- Militari statements under the “inquire” must be enclosed in (“{“…”}”)
- Variable initialization, condition and increment/decrement statements should be always within an inquire loop.
- There could be a multiple initialization of variables inside the enclosed parenthesis of the inquire statement, only variables with integer values are allowed in the initialization part.

SYNTAX: ·

```
inquire(initialization; condition ; increment/decrement) {
<statement>
}
```

SAMPLE:

```
inquire(a=0;a<=5;a++) {
post(a);
}
```

.

XI. STRUCT

Programming Syntax

Structs are used to declare a user-defined data type which groups and stores related values in variables with different data types called as member/s.

RULES:

- Struct declaration and definition must be before any other function in the code.
- Struct or structure must begin and end in “{“ and “}”.
- The values of the struct members must match with its data types.
- All data types are available to be used in struct.
- The rules in naming an identifier also applies in struct statement.
- To access any member of the struct, use the member access operator dot (.) between the struct variable name and the member variable name you want to access.
- The struct declaration and definition must come first before the declaration of struct variables inside the function. Declaration of two or more structure variables can be done but must be seperated by comma (,) and end with a semi colon (“;”).
- To initialize a value to a struct member variable, the struct name, a dot and the struct variable name, after that use a space then the assignment operator (=) followed by the value of the struct member.
- Name of the members in the structures must not be the same in the whole program.

SYNTAX:

```
struct <identifier> {

<members declarations>;

};
```

SAMPLE:

```
struct employee {

unit empnum;

company empname;

unit empage

};


PrimaryMission() {

employee s;


} deploy();
```

XII. FUNCTIONS

Programming Syntax

Function is a group of statements that together perform a task. Every Militari program has at least one function, which is PrimaryMission as main, and all programs may define additional functions.

RULES:

- Miss is the data type used if the function has no return value.
- Functions must be placed before the PrimaryMission() function.
- The parameters of a function can be empty if it doesn’t have a value to be obtained when it is called. The numbers of arguments must be the same as the parameter called in function.
- The data type of the parameter given must be same type. Multiple arguments must be separated by comma (,).
- Only variable and literals are allowed to be passed as arguments.
- In case that the function has called a return value, this can only be assigned to a variable with the same data type as the function of the return value, but it is also possible that the function call is not assigned to a variable.
- Return values are preceded by reserved word **backup.** Return value must match the return type of the function.
- The **backup** word should be placed at the end of each functions.

DECLARING - DEFINING A FUNCTION

In Militari, functions must be placed before the PrimaryMission Function before they can be used. A function declaration and definition tells the compiler about a function name and how to call the function. You can define a function by providing its return value, name, and the types for its arguments.

RULES:

- Functions must be placed before the PrimaryMission function.
- The parameters of a function can be empty if it doesn't have a value to be obtained when it is called.

SYNTAX:

```
<return_type> <identifier> ( <parameter/s> ){
<statement/s>
}
```

Where,

- **Return Type**: A function may return a value. The return type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword *miss*.
- **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

CALLING A FUNCTION

While creating a Militari function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

SYNTAX:

```
<identifier> = <identifier>( <parameter/s> );

<identifier> ( <parameter/s> );
```

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

FUNCTION ARGUMENTS

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, the call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

XIV. REGULAR DEFINITION

Name	Definition
upperChar	{ A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z }
lowerChar	{ a, b, c, d, e, f, g, h, I, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z }
zero	{ 0 }
num	{1, 2, 3, 4, 5, 6, 7, 8, 9}
arithOp	{ +, -, *, / }
logicOp	{ && , || }
RelOp	{ <, >, <=, =>, ==, != }
EqualOp	{ = }
ASCII	Any printable character.
Bool	{ AFFIRMATIVE, NEGATIVE }
delim1	{ (}
delim2	{ upperChar, lowerChar, num, “, (, ‘, space, - }
delim3	{ newline, space }
delim4	{ space, = , ; }
delim5	{ space }
delim6	{ space, { , newline }
delim7	{ lowerChar, upperChar, zero, num, (, space }
delim8	{ lowerChar, upperChar, num, (, space }
delim9	{ lowerChar, upperChar, zero, num, space }
delim10	{ lowerChar, upperChar, num, zero, “, (,), {, }, @, !, ||, && , _ , $, * ,- ,+ ,/ ,> ,< ,? ,. ,[,] ,\, space }

delim11	{ space, newline }
delim12	{ space, +, −, *, /, ;, { }
term1	{ armop, logop, relop, ;,], }, comma, :, space }
term2	{ armop, logop, relop, ;,), }, comma, :, ., space }
term3	{ ;, }, comma, space }
term4	{ ;, }, comma,), space }
term5	{ =, comma, armop, logop, relop, (,), [,], }, ;, space }
term6	{ space, newline }

XIII. RESERVED WORDS

WORD	COUNTERPART	DESCRIPTION
PrimaryMission	main	Main function.
Commence	Clrscr	Clear Screen.
post	printf	Displays output.
capture	scanf	Accepts input.
hold	Const	A constant identifier declaration
backup	Return	Terminates and returns value in a function.
campaign	Switch	Statement body that contains multiple selection
operation	Case	Choices for switching a statement.
abort	Break	Function used to end a loop or a condition.
unit	Int	Data type for integer variable.
joe	Char	Data type for character variable.
digit	Float	Data type for float variable.
company	String	Data type for string variable.
inquire	For	Defining the number of loops of iteration.
inorder	If	Condition used to check the statement is true or false.
otherorder	else if	Condition used to check the second or many statement is it is true or false.
order	Else	Last option if the if statement is false.

go	Do	Process of looping where this reserved is used to do first. Partner of **phase()**
phase	while	The loop iterates phase the condition is true.
miss	Void	Used when the function will not return a value
action	Default	Used as the default catch in a switch function.
Extent	Length	Extent returns the character count in the string instance.
Carry	Contains	This method searches strings. It checks if one substring is contained in another.
ToJoeRange	ToCharArray	Converts strings to character arrays. It is called on a string and returns a new char array.
Swap	Reverse	A string can be reversed.
deploy	Getch	End of the program.

XV. RESERVED SYMBOLS

Reserved Symbols	Regular Expression	Token
+	(+)	+
-	(-)	-
*	(*)	*
/	(/)	/
%	(%)	%
=	(=)	=
;	(;)	;
.	(.)	.
&	(&)	&
||	(|) (|)	||
!	(!)	!
//	(/) (/)	//
++	(+) (+)	++
- -	(-) (-)	--
\n	(\) (n)	\n
\t	(\) (t)	\t
!=	(!) (=)	!=
(	(()	(
)	())	)
“	(“)	“
‘	(‘)	‘
:	(:)	:
{	({)	{
}	(})	}

<	(<)	<
>	(>)	>
<=	(<)　(=)	<=
>=	(>)　(=)	>=

XVI. TRANSITIONAL DIAGRAM