**Specific Rules**

1. ***Constants***

Declaration and Initialization of Constants

1. Constant variables *can only be declared globally*.
2. Declaration of constant variables *must begin with the reserved word* **fixed** (*must always be in lowercase*), followed by a space, a data type, a space & the *initialization* of an identifier.
3. All declarations and initializations *must be terminated by a period* (.).
4. It is *required to initialize constant variables* upon its declaration.
5. Initialization is done by *assigning a value to the identifier* with the use of the assignment operator (=), i.e. **fixed** *<data type> <id>=<value>*.
6. The value to be assigned to an identifier *must be in respect to its data type*.
7. The value of a constant is *the value on its first assignment*. So, if a constant were given or resulting a new value, it will still acknowledge its original value the next time it is used.
8. Constant variables with the same data type can be initialized in a single line as long as the identifiers alongside their values are *separated with a comma* (,), i.e. **fixed** *<data type> <id>=<value>, <id>=<value>, <id>=<value>*.
9. Constant variables with different data types can be initialized in a single line as long as they are *separated with a period* (.), i.e. **fixed** *<data type> <id>=<value>*. **fixed** *<data type> <id>=<value>*. **fixed** *<data type> <id>=<value>*.

|  |  |
| --- | --- |
| Syntax: | Example: |
| fixed *<data type> <id>=<value>*.  fixed *<data type> <id>=<value>, <id>=<value>*.  fixed *<data type> <id>=<value>*. fixed *<data type> <id>=<value>*. | **fixed** **number** a=5.  **fixed** **decimal** b=0.1, c=1.1.  **fixed** **number** a=5. **fixed** **decimal** b=0.1. |

1. ***Variables***

Rules

1. Variables *can be declared and/or initialized globally and locally*.
2. All declarations and initializations *must be terminated by a period* (.).
3. Variables with the same data type can be declared and/or initialized in a single line as long as the identifiers with or without values are *separated with a comma* (,), i.e. *<data type> <id>, <id>, <id>=<value>*.
4. Variables with different data types can be declared and/or initialized in a single line as long as they are *separated with a period* (.), i.e. *<data type> <id>. <data type> <id>. <data type> <id>=<value>.*

Declaration of Variables

1. Declaration of variables *must begin with a data type*, followed by a space, and an identifier.

Initialization of Variables

* 1. Initialization is done by *assigning a value to the identifier* with the use of the assignment operator (=), i.e. *<data type> <id>=<value>*.
  2. The value to be assigned to an identifier *must be in respect to its data type*.
  3. The value of a variable is *the value on its* *most recent assignment*. So, if a variable were given or resulting a new value, it will disregard its original value.

|  |  |
| --- | --- |
| Syntax: | Example: |
| *<data type> <id>.*  *<data type> <id>, <id>, <id>=<value>.*  *<data type> <id>. <data type> <id>.* | **number** a.  **decimal** b, c, d=1.32.  **number** a. **decimal** b. |

1. ***Data Types and Literals***

A *data type* is a classification of the type of data that a variable or object can hold in computer programming. Here’s the list of data types a ROOK programmer can use:

|  |  |  |  |
| --- | --- | --- | --- |
| C Language | C++ Language | Proposed Language | Description |
| int | int | number | A data type that holds the value of whole numbers. |
| float | float | decimal | A data type that holds the value of decimal numbers. |
| string | string | string | A data type that holds the value of characters. |
| bool | bool | boolean | A data type that holds the value of ‘true’ and ‘false’. |

In contrast, a variable is a name that can represent different values during the execution of the program. And a constant is a name that represents the same value throughout the program. But a *literal* is not a name ― it is the value itself.

1. *Number* – a data type that holds the value of whole numbers.

Rules for Number Data Type

* 1. The data type **number** is only used when declaring an *integer type variable*.
  2. It *only accepts whole numbers*.
  3. The value to be assigned to its identifier *must be a number literal*.
  4. The reserved word **number** *must always be in lowercase letters*.
  5. It *must always* be followed by a space.

Rules for Number Literal

1. The value of a numberliteral *can only range from -999999999 to 999999999*, otherwise it will not be accepted.
2. Leading zeros *will be omitted*, i.e. 007 will be treated as 7.
3. The *tilde sign (~)* is be used to indicate a negative value.
4. When assigning a number literal to an identifier with a number data type, *simply put the value* next to the assignment operator (=), i.e. **number** <id>=<number literal>.

|  |  |  |
| --- | --- | --- |
| Valid Number Literal | Invalid Number Literal | Reason |
| 12345 | “12345” | Enclosed with “” |
| 123 | ‘123’ | Enclosed with ‘’ |
| ~123 | Yes | Character value |
| 87000 | 87000.01 | Decimal value |
| 123456789 | 12#$%^&89 | Special characters |
| *999999999* | *99999999999999999999* | Value exceeded |

1. *Decimal* **–** a data type that holds the value of decimal numbers.

Rules for Decimal Data Type

1. The data type **decimal** is only used when declaring a *floating type variable*.
2. It *only accepts decimal numbers*.
3. The value to be assigned to its identifier *must be a decimal literal*.
4. The reserved word **decimal** *must always be in lowercase letters*.
5. It *must always* be followed by a space.

Rules for Decimal Literal

1. The value of a decimalliteral *can only range from -999999999.99999 to 999999999.99999*, otherwise it will not be accepted.
2. Trailing zeros to the right of a decimal point *will be omitted*, i.e. 2.50 will be treated as 2.5.
3. The *tilde sign (~)* is be used to indicate a negative value.
4. When assigning a decimal literal to an identifier with a decimal data type, *simply put the value* next to the assignment operator (=), i.e. **decimal** <id>=<decimal literal>.

|  |  |  |
| --- | --- | --- |
| Valid Decimal Literal | Invalid Decimal Literal | Reason |
| 12345.67 | “12345.67” | Enclosed with “” |
| ~123.45 | Yes | Character value |
| 1234.56 | 1234 | Number value |
| 1234567.89 | 12#$%^&.89 | Special characters |
| *9999999999.99* | *9999999999999999999999.99* | Value exceeded |

1. *String* – a data type that holds the value of characters.

Rules for String Data Type

1. The data type **string** is only used when declaring a *string and character type variable*.
2. It *only accepts a finite sequence of characters*.
3. The value to be assigned to its identifier *must be a string literal*.
4. The reserved word **string** *must always be in lowercase letters*.
5. It *must always* be followed by a space.

Rules for String Literal

* 1. The value of a string literal *can be* *any printable character*.

1. When assigning a string literal to an identifier with a string data type, the value *must be placed after the assignment operator (=)* and it *must be enclosed in a double quotation mark (“”)*, i.e. **string** <id>=<“string literal”>.
2. If the string literal is a double quotation mark, a *backslash (\) is required* to be placed at the end of the said character before enclosing it with a double quotation mark (“”), i.e. ““Hello”\”.

|  |  |  |
| --- | --- | --- |
| Valid String Literal | Invalid String Literal | Reason |
| “AEIOU” | “AEIO”U | Character is outside the “” |
| “Automata” | ‘Automata’ | Not enclosed in “” |
| “Compil3r” | Compil3r | Not enclosed in “” |
| “123” | 123 | Number value |
| “123.45” | 123.45 | Decimal value |

1. *Boolean* – a data type that holds the value of ‘true’ and ‘false’.

Rules for Boolean Data Type

* + 1. The data type **boolean** is only used when declaring a *boolean type variable*.
    2. It *only accepts the values true and false*.
    3. The value to be assigned to its identifier *must be a boolean literal*.
    4. The reserved word **boolean** *must always be in lowercase letters*.
    5. It *must always* be followed by a space.

Rules for Boolean Literal

1. The value of a boolean literal *can only be a true or a false*, otherwise it will not be accepted.
2. The boolean literals *must always be in lowercase letters*.
3. When assigning a boolean literal to an identifier with a boolean data type, *simply put the value* next to the assignment operator (=), i.e. **boolean** <id>=<boolean literal>.

|  |  |  |
| --- | --- | --- |
| Valid Boolean Literal | Invalid Boolean Literal | Reason |
| true | “true” | Enclosed with “” |
| false | ‘false’ | Enclosed with ‘’ |
| true | 123 | Number value |
| false | 123.45 | Decimal value |
| true | 12#$%^&89 | Special characters |
| false | fALsE | Toggle case letters |
| true | TRUE | Uppercase letters |

1. ***Identifiers***

Rules in Naming Identifiers

1. It *must always start with a small letter* followed by *any alphanumeric character*.
2. *Use of any special characters* such as underscore, space, dash, etc. *is prohibited*.
3. 1 up to 10 *alphanumeric characters only* are allowed.
4. Identifiers are *case sensitive*. E.g. *rOoK* and *rook* are two different identifiers.
5. Reserved words are *not allowed* to be used as identifiers.

|  |  |  |
| --- | --- | --- |
| Valid Identifier | Invalid Identifier | Reason |
| area | Area | Starts with a capital letter |
| skill1 | 1skill | Starts with a number |
| question | ?question | Starts with a special character |
| perimeterTri | perimeter\_tri | Use of special character |
| studentNum | studentNumber | More than 10 alphanumeric characters |
| fix | fixed | Use of reserved word |

1. ***Whitespace Characters***

ROOK Compiler ignores whitespace characters such as spaces, line feeds, indentions, etc. However, there must be at least one space between words that you intend to separate.

|  |
| --- |
| Example: |
| fixed number=5.  file structure.  do a1+1. |

In expressions, whitespace characters are not necessary. There can be no spaces between an operator and identifier.

|  |
| --- |
| Example: |
| number x=0.  do a++.  c+1.  5+4. |

Also, part of whitespace characters are comments. You can put comments using slash “/” followed by another slash “/”. Multiline line comment is not supported on ROOK compiler. Note that the whole line will be a comment.

|  |
| --- |
| Example: |
| // This is a comment |

1. ***Operators***

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. ROOK provides the following type of operators:

|  |  |
| --- | --- |
| * Arithmetic Operator * Assignment Operator * Relational Operator | * Logical Operator * Unary Operator |

1. *Arithmetic Operators*

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| + | Addition | Adds two operands. |
| - | Subtraction | Subtracts second operand from the first. |
| \* | Multiplication | Multiplies both operands. |
| / | Division | Divides numerator by de-numerator. |
| % | Modulus | Yields the remainder after integer division. |

1. *Assignment Operators*

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| = | Simple assignment operator | Assigns values from right side operands to left side operand |
| += | Add AND assignment operator | It adds right operand to the left operand and assign the result to left operand |
| -= | Subtract AND assignment operator | It subtracts right operand from the left operand and assign the result to left operand |
| \*= | Multiply AND assignment operator | It multiplies right operand with the left operand and assign the result to left operand |
| /= | Divide AND assignment operator | It divides left operand with the right operand and assign the result to left operand |

1. *Relational Operators*

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| == | Equal to | Checks if the values of two operands are equal or not, if yes then condition becomes true. |
| != | Not equal to | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. |
| > | Greater than | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| < | Less than | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. |
| >= | Greater than or equal | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. |
| <= | Less than or equal | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. |

1. *Logical Operators*

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| and | Logical AND | If both operands are true, the condition will be true. |
| or | Logical OR | If any operand is true, the condition will be true. |
| ! | Logical NOT | Used to reverse the current state of the operand. |

Rules

1. Logical Operators *can only be used in conditional and looping statements*. It is used in comparing boolean expressions.
2. The reserved words **and** and **or** *must always be in lowercase letters*.
3. *Unary Operators*

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| ++ | Increment | Increases integer value by one. |
| -- | Decrement | Decreases integer value by one. |

Rules

1. Unary Operators *can only be used for integer type variables*.

Increment

1. *Pre-Increment* – used to increment their operand by 1, and the value of the expression is the resulting incremented value.

|  |  |
| --- | --- |
| ++a | Assumes that a is an integer |

1. *Post-Increment* – used to increment the value of their operand by 1, but the value of the expression is the operand’s original value prior to the increment operation.

|  |  |
| --- | --- |
| a++ | Assumes that a is an integer |

Decrement

1. *Pre-Decrement* – used to decrement their operand by 1, and the value of the expression is the resulting decremented value.

|  |  |
| --- | --- |
| --a | Assumes that a is an integer |

1. *Post-Decrement* – used to decrement the value of their operand by 1, but the value of the expression is the operand’s original value prior to the decrement operation.

|  |  |
| --- | --- |
| a-- | Assumes that a is an integer |

1. ***Operators Precedence***

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator | Description | Associativity |
| 1 | ++  --  ( )  [ ]  @ | Suffix Increment  Suffix Decrement  Function Call  Subscript  Member Access | Left-to-Right |
| 2 | ++  --  ! | Prefix Increment  Prefix Decrement  Logical NOT | Right-to-Left |
| 3 | \*  /  % | Multiplication  Division  Modulus | Left-to-Right |
| 4 | +  - | Addition  Subtraction | Left-to-Right |
| 5 | >  <  >=  <= | Greater than  Less than  Greater than or equal  Less than or equal | Left-to-Right |
| 6 | ==  != | Equal to  Not equal to | Left-to-Right |
| 7 | and | Logical AND | Left-to-Right |
| 8 | or | Logical OR | Left-to-Right |
| 9 | =  +=  -=  \*=  /= | Simple assignment  Add AND assignment  Subtract AND assignment  Multiply AND assignment  Divide AND Assignment | Right-to-Left |
| 10 | , | Comma | Left-to-Right |

1. ***Expression***

An expression is a combination of one or more explicit values, constants, variables, and operators, that the programming language interprets and computes to produce another value.

Rules

1. An expression *must always consist of operator/s and operand/s*.
2. It *must always start with an operand*, followed by an operator and an operand, or a combination of one or more values, constants, variables, and operators.
3. An operand *cannot be* followed by another operand.
4. An operator *cannot be* followed by another operator.
5. Operators *should be in between* constants, variables, operands, and/or expressions.
6. An arithmetic expression *must be written in straight-line form*, otherwise it will not be considered as an expression.
7. Parentheses are used in the *same manner as in algebraic expressions*, i.e. PEMDAS.
8. The *tilde sign (~)* is used to indicate a negative value.

|  |  |  |
| --- | --- | --- |
| Valid Expression | Invalid | Reason |
| a+b | +a+b | Doesn’t start with an operand |
| c+d | c+ | Operator is not followed by an operand |
| e+f+g | ef+g | Operand followed by another operand |
| h+i+j | h+-i+j | Operator followed by another operator |

1. ***Statements***
2. *Assignment Statements* – sets and/or re-sets the value stored in the storage location(s) denoted by a variable name. In other words, it copies a value into the variable.

|  |
| --- |
| Syntax: |
| *<id> = <value>.*  *<id> += <expression>.*  *<id> -= <value>.* |

Rules

1. Assignment statements *must always* start with an identifier, which is followed by an assignment operator, and then a value or expression.
2. The variable on the left hand side of the assignment operator *will have the value given by the expression on the right hand side*.
3. The data type of the variable on the left hand side *should be the same* as the data type returned by the expression.
4. If the data type of the variables on the left and right side differs, *the result will be converted to the data type of the variable on the left hand side*.
5. Once the assignment statement takes place, the *original value of the variable will be disregarded*.
6. Assignment statements *must be terminated by a period (.)*.
7. *Input and Output Statements* – a portion of a program that instructs a computer how to read and process information. It pertains to gather information from an input device, or sending information to an output device.

*Write statement* is used to send formatted output to the standard output (screen).

|  |
| --- |
| Syntax: |
| *write <statement>.* |

Rules

1. The reserved word **write** *must always be in lowercase letters*.
2. To *display the value of a variable*, it must begin with the reserved word **write**, followed by a space, and the variable. A comma (,) must be used to display multiple variables in one statement.
3. To *display a string*, it must begin with the reserved word **write**, followed by a space, and the string, which is enclosed with a double quotation mark (“”).
4. To *display a combination of the value of a variable and a string*, it must begin with the reserved word **write**, followed by a space, and then separate the variable and the string with a comma (,).
5. Write statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| write a.  write a, b, c, d.  write “Hello World”.  write “Hello”, a, “World”, b. |

*Read statement* is used to read formatted input from standard input (keyboard).

|  |
| --- |
| Syntax: |
| *read <statement>.* |

Rules

1. The reserved word **read** *must always be in lowercase letters*.
2. To *read an input*, it must begin with the reserved word write, followed by a space, and the identifier, which will hold the value of the input.
3. The *valid holders of the value of an input* are variables, array’s elements, and object class of a file.
4. There *must be one read statement for every input*.
5. Read statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| read a.  read array[0].  read Artist@Movies. |

1. *Conditional Statements* – specify whether another statement or block of statements should be executed or not.

*If Statements (If, ElseIf, Else)* – executes a statement or block of statements if a specified condition is true or false.

|  |
| --- |
| Syntax: |
| *if (<boolean\_expression>)*  *start*  *<statement>.*  *end.*  *elseif ((<boolean\_expression>) or (<boolean\_expression>))*  *start*  *<statement>.*  *end.*  *elseif ((<boolean\_expression>) and (<boolean\_expression>))*  *start*  *<statement>.*  *end.*  *else*  *start*  *<statement>.*  *end.* |

Rules

1. The reserved words **if**, **elseif**, **else**, **start**, and **end** *must always be in lowercase letters*.
2. In using if statements, it *must always* begin with the reserved word **if**, it may be followed by a space, and then the boolean expression, which is enclosed with a parenthesis (()). After that, a statement or a block of statements must be placed and it must be enclosed using the reserved words **start** and **end**, respectively, and it *must be terminated by a period (.)*. Otherwise, it will not be accepted. Same rules are applied into using **elseif**.
3. There *must always* be a boolean expression inside the parenthesis after the keyword **if** and/or **elseif**.
4. Logical operators *should be used* when there are multiple boolean expressions.
5. If statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| if (a==5)  start  write “a is equal to 5”.  a++.  end.  elseif ((a<4)or(<a==4>))  start  write “a is less than 5”.  end.  elseif ((a>5)and(a<50))  start  write “a is greater than 5 but less than 50”.  end.  else  start  write “a is greater than 50”.  end. |

*Switch-Choice Statements* – allows a variable to be tested for equality against a list of values

|  |
| --- |
| Syntax: |
| *switch(<expression>)*  *start*  *choice <constant-expression>: <statement>. stop.*  *choice <constant-expression >: <statement>. stop.*  *choice <constant-expression >: <statement>. stop.*  *default: <statement>. stop.*  *end.* |

Rules

1. The reserved words **switch**, **choice**, **default**, **stop**, **start**, and **end** *must always be in lowercase letters*.
2. In using switch-choice statements, it *must always* begin with the reserved word **switch**, it may be followed by a space, and then the expression which is enclosed with a parenthesis (()). Otherwise, it will not be accepted.
3. The *only data types which are accepted by switch-choice statements* are **number** and **string**.
4. The reserved words **start** and **end** are *required* to indicate the body of **switch**.
5. The body of **switch** *must always* have the reserved word **choice**, and should be followed by a space, and a constant expression which ends with a colon (:).
6. If the constant expression is a string, it *must be enclosed with a quotation mark (“”)*.
7. A statement or a block of statements *should be placed* after the colon in a choice statement.
8. It is *required* to have the reserved word **stop** right after a statement or a block of statements in every **choice**.
9. There *must always* be a **default** for switch-choice statements. It is used when the expression to be tested did not satisfy the condition in every **choice**. Same rules with **choice** are applied.
10. Switch-choice statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| switch(a)  start  choice 5: Write “a is equals to 5”. stop.  choice 10: Write “a is equals to 10”. stop.  default: Write “I don’t know”. stop.  end. |

1. *Looping Statements* – allows a statement or group of statements to be executed multiple times.

*For Statement* – executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

|  |
| --- |
| Syntax: |
| *for (<initialization>,<condition>,<iteration>)*  *start*  *<statement>.*  *………*  *end.* |

Rules

1. The reserved words **for**, **start**, and **end** *must always be in lowercase letters*.
2. For statements must always begin with the reserved word **for** and it should be followed by an initialization, condition, and iteration, all separated by a comma (,), and then enclose the three in a parenthesis (()).
3. The reserved words **start** and **end** are required to indicate the body of **for**.
4. A statement or a block of statements should be placed inside the body of **for**.
5. Nested looping statements are allowed.
6. For statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| for (number a=0, a<10, a++)  start  write a.  newline.  end. |

*While Statement* – repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

|  |
| --- |
| Syntax: |
| *while (<condition>)*  *start*  *<statement>.*  *……….*  *end.* |

Rules

1. The reserved words **while**, **start**, and **end** *must always be in lowercase letters*.
2. While statements must always begin with the reserved word **while** and it should be followed by a condition which is enclosed in a parenthesis (()).
3. The reserved words **start** and **end** are required to indicate the body of **while**.
4. A statement or a block of statements should be placed inside the body of **while**.
5. Nested looping statements are allowed.
6. While statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| while (a<10)  start  Write a.  Newline.  a++.  end. |

*Do-While Statement* – like a while statement, except that it tests the condition at the end of the loop body.

|  |
| --- |
| Syntax: |
| *do*  *start*  *<statement>.*  *……….*  *end.*  *while (<condition>).* |

Rules

1. The reserved words **do**, **while**, **start**, and **end** *must always be in lowercase letters*.
2. The reserved word **do** should always be the first to be stated before the **while** statement, otherwise it will not be accepted.
3. The reserved words **start** and **end** are required to indicate the body of **do**.
4. A statement or a block of statements should be placed inside the body of **do**.
5. The reserved word **while** should be followed by a condition which is enclosed in a parenthesis (()).
6. Nested looping statements are allowed.
7. Do-while statements *must be terminated by a period (.)*.

|  |
| --- |
| Example: |
| do  start  write a.  newline.  a++  end.  while (a<10). |

1. ***Functions***

Rules in Making a Function

1. The reserved words **null**, **start**, and **end** *must always be in lowercase letters* while **!START** and **END!** must always be in uppercase.
2. A program must have a *main function* (**!START**) which the operating system calls when the end-user runs the program.
3. The *main function* has its own terminator― **END!**.
4. Functions, *with the exception of the main function*, must have a **start** and an **end**. It indicates the body of a function.
5. Functions *must have at least one statement* placed inside its body.
6. Other functions, *must be placed before the main function* (**!START-END!**).
7. The data type that will be returned by a function *must be specified*. Use **null** if the function has *no value* to return.
8. If a function is not null, it *must have a return statement*.
9. Functions with parameter are *required to declare variables*. It must be enclosed with a parenthesis (()) following the name of the function. Use comma (,) to separate multiple variables.

Rules for Calling a Function

1. Functions are called through its name followed by a parenthesis (()).
2. If the function to be called has a parameter, the parameter must be placed inside the parenthesis (()).
3. Statements which calls a function are treated the same way as other statements. Therefore, it is also terminated by a period (.).
4. Functions are allowed to call itself.
5. When the function called corresponds to a variable, the data type should be the same for both function and variable.

|  |
| --- |
| Syntax: MAIN FUNCTION (!START) |
| *!START*  *<statement1>.*  *<statement2>.*  *<statement3>.*  *END!* |

|  |
| --- |
| Syntax: FUNCTION |
| *<data type> <function name> (<data\_type> <variable1>)*  *start*  *<statement1>.*  *<statement2>.*  *<statement3>.*  *return <variable1>.*  *end.* |

|  |
| --- |
| Example: FUNCTION |
| number addfive(number p1)  start  return p1+5.  end. |

|  |
| --- |
| Syntax: CALLING A FUNCTION |
| *variable2* = *function\_name(variable1).* |

|  |
| --- |
| Example: CALLING A FUNCTION |
| num1 = addfive(sample1). |

1. ***Array***

Rules for Array Declaration

1. Arrays *can be declared globally or locally*.
2. The declaration of arrays is *same with the declaration of variables*, except you need to include the size of the array using square brackets.
3. The size of an array *must always be a positive whole number*.
4. The *minimum size of an array is 1*, and the *maximum is 50*.
5. Changing the size of an array once it is declared is *not allowed*.
6. The data type of the element which will be stored in the array *must be the same* as the data type of the array.
7. The declaration of multiple arrays in one statement is *allowed*.
8. The declaration of multidimensional array is *allowed*.
9. The *maximum dimension of an array is 2*.
10. All statements, including array declarations is *terminated by a period (.)*.
11. The initialization of the value of an array *can be done* in its declaration.
12. The number of initialized value *can be less than* the size of the array, *but not greater than*.

Rules in Using Array

1. An array always begins with index 0.
2. Using an array more than the size you declared is not allowed.

|  |
| --- |
| Syntax: |
| *<data type> <variable1>[array size], <variable2>[array size].*  *<data type> <variable3>[array size] = {value1,value2}.*  *<data type> <variable4>[array size] = {{value1,value2},{value3,value4}}.* |

|  |
| --- |
| Example: |
| number a[25], b[10], c[10] = {3,1,4,3,1,4,2,3,1,5}.  decimal d[2][2], e[5][5], f[2][2] = {{ 0,1 },{3,2}}. |

1. ***File***

Rules for File Declaration

* + 1. The reserved words **file**, **start**, and **end** *must always be in lowercase letters*.
    2. File *can only be declared globally*.
    3. File is declared by the use of the reserved word **file**, followed by its identifier.
    4. All file declaration must have at least one element.
    5. Each element must have its data type and identifier.
    6. All elements are placed inside the reserved words **start** and **end**.
    7. Initialization of an element inside the file declaration is prohibited.
    8. File variables can be declared after the reserved word **end**, and the terminator period (.). It can also be declared in the main function i.e. **file** <id> <variable>.
    9. The declaration of multiple file variables is allowed. Use comma (,) to separate multiple variables.
    10. The declaration of a file inside a file is allowed. However, the file must be declared first before using it in another file.

Rules in Using File

1. Files are called through its variable name, followed by “@”, and the name of the element.
2. Files are used as a parameter of a function by using the data type of the element as its file name.

|  |
| --- |
| Syntax: FILE DECLARATION |
| *file <file\_name>*  *start*  *<data type> <element>*  *end. <variable1>,<variable2>.* |

|  |
| --- |
| Syntax: CALLING A FILE DECLARATION |
| *<variable>@<element>* |

|  |
| --- |
| Syntax: USING FILE IN A FUNCTION |
| *<data\_type> <function name>(<file name><variable>)*  *start*  *write <variable>@<element>.*  *end* |

|  |
| --- |
| Example: FILE DECLARATION |
| file product  start  number price.  end. banana,apple. |

|  |
| --- |
| Example: CALLING A FILE DECLARATION |
| banana@price |

|  |
| --- |
| Example: USING FILE IN A FUNCTION |
| null wrt(product pr)  start  write pr@price.  end |