|  |
| --- |
| **ROOK: Specific Rules** |

1. **Identifiers**

**I. ROOK rules in naming identifiers**

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

***Example***

|  |  |  |
| --- | --- | --- |
| Valid Identifier | Invalid Identifier | Reason |
| area1 | area\_1 | Use of “\_” |
| value1 | value 1 | Use of “space” |
| studentNum | studentNumber | More than 10 alphanumeric characters Uppercase letter for the first letter |
| schoolYear | Sch-Year | Use of “-“ & first letter is capital |
| delim1 | 1delim | Start with a number |
| digit1234 | 1234\_digit | Start with a number  Use of “\_” |
| question1 | ??Question1 | Start with a special character |

1. **Whitespaces**

ROOK Compiler ignores whitespaces such as spaces, new lines and indents. However, there must be at least one space between words to separate them.

***Example***

FIXED Number = 5.

FILE structure.

DO a1+1.

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

***Example***

Int x=0.

Do a++.

c+1.

5+4.

Also part of whitespaces are comments. You can put comments on ROOK using the less than symbol “<” and closing it with a slash “/” and a greater than symbol “>”.

***Example***

<This is a comment/>

<This is a multi

Line comment/>

1. **Variables**

***Declaration of Variables***

1. It must all begin with a small letter.
2. Variables can be declared globally and locally.
3. All declarations must be terminated by a *period* “.”.
4. Variables with the same data type can be declared in a single line as long as the identifiers with or without values are separated with a *comma* “**,**”.
5. Variables with different data types can be declared in a single line as long as the identifiers with or without values are separated with a *period* “**.**”.

***Initialization of Variables***

* 1. Initialization of variables can be done globally or locally.
  2. The value of a variable is the value of its most recent assignment.
  3. Initialization is done by adding “=”after the declaration of the variable followed by its initial value.
  4. All initializations must be terminated by a period.
  5. Variables with the same data type can be initialized in a single line as long as the identifiers with or without values are separated with a *comma* “,”.
  6. Variables with different data types can be initialized in a single line as long as the identifiers with or without values are separated with a *period “.”*.
  7. The value of an initialized variable must be in respect to its data type.

<data type> <id>.

<data type> <id>,<id>,<id>=<value>.

***Syntax:***

number a.

decimal b,c,d=1.32.

***Example***

1. **Constants**

***Declaration of Constants***

* 1. Constant variables can only be declared globally.
  2. The value of a constant is the value on its first assignment.
  3. Declaration must begin with the keyword **Fixed** followed by the data type.
  4. Initialization is done by adding a character “=” after the declaration of the variable followed by its initial value.
  5. All initializations must be terminated by a period.
  6. Constants with the same data type can be initialized in a single line as long as the identifiers with or without values are separated with a comma “**,**”.
  7. Constants with different data types can be initialized in a single line as long as the identifiers with or without values are separated with a *period* “**.**”.
  8. The value of an initialized variable must be in respect to its data type.
  9. It is required to initialized a constant upon it’s declaration.

***Sample***

1. To declare or initialize a constant variable:

***Syntax:***

Fixed *<data type> <id>=<value>.*

Fixed *<data type> <id>=<value>,<id>=<value>.*

Fixed number a=5.

Fixed decimal b=0.1,c=1.1.

***Example***

1. **Data Types and Literals**

***Data types*** defines the values it can take and the operations that can be performed on it. *Here’s the list of data types a ROOK programmer can use:*

|  |  |  |  |
| --- | --- | --- | --- |
| C Language | C++ Language | Proposed Language | Description |
| String | String | String | A data type that holds multiple characters. |
| Bool | Bool | Boolean | A data type that can only hold the value of ‘true’ or ‘false’. |
| int | int | Number | A data type that that can store Integers. |
| float | float | Decimal | A data type that will hold a value of numbers with decimals. |

A value written exactly as it's meant to be interpreted. 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 a program. But a ***literal*** is not a name -- it is the value itself.

1. Value of an Number literal can range from ***-999999999*** to ***999999999***. Exceeded to this range is not valid.
2. Leading zeros will be disregarded. *Number* will be valid but the fractional part will be omitted.
3. **Number**- A data type that contains whole numbers.

***Rules for Number Data Type***

* 1. The data type *Number* can be used in declaring an integer type variable.
  2. Accepts whole numbers only.
  3. Input of the data type should be a Number literal.

|  |  |  |
| --- | --- | --- |
| VALID NUMBER DATA TYPE | INVALID NUMBER DATA TYPE | REASON |
| Number a | Number 1 | Invalid variable |
| NUmber aBC | Number | No variable |
| NumbeR value = 10 | Number value = 1.50 | Decimal value |
| Number num1=5 | Number num1=’a’ | Character value |

***Rules for Number Literals***

|  |  |  |
| --- | --- | --- |
| Valid Number Literal | Invalid Number Literal | Reason |
| 12345 | “12345” | Enclosed with “” |
| 123 | ‘123’ | Enclosed with ‘’ |
| -123 | Yes | Letter |
| 12345678 | 12&@!#56789 | Special Characters |
| *999999999* | *99999999999999999999* | Value exceeded |

1. **Decimal-** A data type that contains whole numbers and a decimal part. (64-bit)

***Rules for Decimal Data Type***

1. The data type *Decimal* can be used in declaring a floating type variable.
2. When initializing, the special character “=” followed by its initial value must be appended after the identifier.
3. Accepts numbers with decimal point.
4. Input of the data type should be a *Decimal* literal.

|  |  |  |
| --- | --- | --- |
| Valid Double Data Type | Invalid Double Data Type | Reason |
| Decimal num. | Decimal 1.50. | Invalid variable |
| Decimal num = 10.50 | Decimal = 10. | Whole number initialized |
| Decimal schoOlYeAr = 2017.2018 | Decimal schoolYear = 2017.2018.50. | Used of 2 periods |
| Decimal value=2.5069. | Decimal Value=”2.50” | Enclosed in “” & start with capitalized letter |
| Decimal digit=1.00 | Decimal digit=1,00 | Invalid value initialized |

***Rules for Decimal Literals***

1. Value of a *Decimal* literal can range from ***-999999999.99999*** to ***999999999.99999***. Exceeded to this range is not valid.
2. Trailing zeros will be disregarded after “.” If exceeded 4

|  |  |  |
| --- | --- | --- |
| Valid Decimal Literal | Invalid Decimal Literal | Reason |
| 12345.50 | “12345.50” | Enclosed with “” |
| -123.21 | Yes | Letter |
| 12345678.06 | 12&@!#56789.06 | Special Characters |
| *9999999999.99* | *9999999999999999999999.99* | Value exceeded |

1. ***String-*** A data type that represent a series of characters*.*

***Rules for String Data Type***

1. The data type *String* can be used in declaring a string type variable.
2. When initializing, the special character “=” followed by its initial value must be appended after the identifier.
3. In initializing its value, the character must be enclosed in a double quote “”””.
4. Initializing a double quote inside a string requires a “\” then the double quotaion.

|  |  |  |
| --- | --- | --- |
| Valid String Data Type | Invalid String Data Type | Reason |
| String school. | String 1234 | Invalid variable |
| String motto = “L0v3 i5 l1Ke A ROS4Ry FulL of MisTer1” | String Motto = Love is like a rosary full of mystery. | Not enclosed in “” |
| String slogan = “Make IT Happen” | String – “Make it happen” | Use of “-“ instead of “=” |

***Rules for String Literal***

|  |  |  |
| --- | --- | --- |
| Valid String Literal | Invalid String Literal | Reason |
| “AEIOU” | “AEIO”U | Syntax |
| “Automata” | ‘Automata’ | Not enclosed in “” |
| “Compil3r” | Compiler | Not enclosed in “” |
| “1” | 1 | Number value |
| “ROOK” | “Rook’ | Inconsistency in ‘’ |

1. Value of a *String* literal can be any printable character.
2. *String* literals should be enclosed by a double quote “ **“ ”** ”.

**D. *Boolean-*** A data type can only handle two values: true or false*.*

***Rules for Boolean Data Type***

* + 1. The data type *Boolean* can be used in declaring a *Boolean* type variable.
    2. There are only 2 valid values: true or false.
    3. When initializing, the special character “=” followed by its initial value must be appended after the identifier.
    4. Input of the data type should be a *Boolean* literal.

|  |  |  |
| --- | --- | --- |
| Valid Boolean Data type | Invalid Boolean Data type | Reason |
| Boolean answer | Boolean answer = 1 | Invalid value |
| Boolean yes = true | Boolean yes = “true” | enclosed in “” |
| Boolean no = true | Boolean no = ‘false’ | enclosed in ‘’ |

|  |  |  |
| --- | --- | --- |
| Valid Boolean Literal | Invalid Boolean Literal | Reason |
| True | T | Single letter |
| False | Flse | Incomplete |

***Rules for Boolean Literals***

* 1. Value of a *Boolean* literal can only be a *True* or a *False*. Otherwise, it is not valid.

1. **Array**

***Rules for Array Declaration***

1. Declaration of Arrays can be declared globally or locally.
2. Declaration of Arrays is the same with declaring variables, but in Array, you need to include the size of the array besides the name of it using square brackets.
3. Declaration of multidimension is allowed.
4. The size of the Arrays should be a positive number.
5. The minimum size of an array is 1 and the largest is 50.
6. The maximum dimension of an array is 2.
7. All statements including array declarations must end with a period.
8. Declaration of multiple arrays in one statement is possible.
9. Initializing the content of an array on its declaration is allowed.
10. Initialization can be less than the size of the array but not greater than.
11. The data type that you will stored in the array must be the same as the data type of the array.
12. Changing the size of the array, once it is declared is not possible.

***Rules for using Array***

1. The start of an array is index 0.
2. You cannot use more than the size you declared.

***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*}}.

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}}.

***Example:***

1. **Functions**

***Rules for making a Function***

1. Every program must have a !Start Function, it serves as the lead function.
2. Every !Start function must have End. this will serves as the terminator of the !Start function.
3. Other functions must be placed before the !Start function.
4. In making a function, the data type that it will return must be specified. Use **Null** if there’s nothing to return.
5. If a function is not null, it should have a **Return** statement.
6. In making a function with a parameter, declare the variables and enclosed it with a parenthesis after the function’s name. Use comma (,) to separate variables.
7. Every function must have **Start** and **End.**
8. Every function must have at least one statement.

***Rules for calling a Function***

1. Functions are called through its name followed by a parenthesis. If a function has a parameter, it is also needed to provide variables or values inside the parenthesis.
2. Like any other statements, period is needed to terminate the statement when a function is called.
3. Functions can call itself.
4. When a function that corresponded to a variable called, the data type should be the same for both function and variable

!START

<*statement1>.*

*<statement2>.*

*<statement3>.*

END.

***Syntax:*!START FUNCTION**

*<data type> <function name> (<data\_type> <variable1>)*

START

**<***Statement1>.*

*<Statement2>.*

*<Statement3>.*

*return <variable1>.*

END.

***Syntax:* FUNCTION**

intaddfive*(*intp1*)*

START

return p1+5.

END.

***Sample:* FUNCTION**

*variable2* = *function\_name(variable1).*

***Syntax:* CALLING A FUNCTION**

Num1 = addfive(sample1).

***Sample:* CALLING A FUNCTION**

1. **File**

***Rules for File Declaration***

* + 1. Declaration of File must be globally.
    2. In declaring, use the keyword File followed by its identifier. Enclosed all the elements with keywords Start and End.
    3. Each element must have its data type and identifier.
    4. Every File must have at least one element.
    5. Initialization of element inside the File declaration is prohibited.
    6. Declaring of File variable must be after the keyword End and terminator period “.”.
    7. Multiple declaring of File variable is allowed, use comma “,” to separate variables.
    8. Declaration of a File inside a File is allowed. However, the file must be declared first before using on another File.

***Rules for using File***

* + 1. On calling a File, it can be access through calling the File variable followed by “@” and the elements name.
    2. On using it as a parameter in a Function, the elements data type will be the File name.

***Syntax: (Declaring)***

File *<file\_name>*

START

*<data type> element>*

END. <variable1>,<variable2>.

***Syntax: (Calling)***

Null wrt(product pr)

START

WRITE pr@price.

END

***Sample: (Using in Function)***

banana@price.

***Sample: (Calling)***

Fileproduct

START

Number price.

END. banana,apple.

***Sample: (Declaring)***

*<data\_type> <function name>(<file name><variable>)*

START

WRITE *<variable>@<element>.*

END

***Syntax: (Using in Function)***

*<variable1>@<element>*

1. **Operators**

An operator is an object that can manipulate a value of operator. ROOK provides the following types of operators:

1. Arithmetic Operators
2. Unary Operators
3. Relational Operators
4. Logical Operators
5. Assignment Operators
6. **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 | Modulus Operator and remainder of after an integer division |

***Rules:***

1. Arithmetic Operators should NOT be next to each other, unless the value after the arithmetic operator is a negative number, then the position of arithmetic operator that is followed by the negative sign and the value will be allowed.
2. Arithmetic Operators should be in the middle of a variable, value or arithmetic operations.
3. **Unary Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| ++ | Increment | increases integer value by one |
| -- | Decrement | decreases integer value by one |

***Rules:***

1. Unary Operators only works on integers.
2. It can be used before or after the integer.

***Increment***

1. **Pre- Increment** – used to increment the value of variable before using in the expression.

++a *Assumes that a is an integer*

1. **Post-Increment** – used to increment the value of variable as soon as after executing expression completely in which post-increment is used.

a++ *Assumes that a is an integer*

***Decrement***

1. **Pre- Decrement** – used to decrement the value of variable before using in the expression.

--a *Assumes that a is an integer*

1. **Post-Decrement** – used to decrement the value of variable as soon as after executing expression completely in which post-increment is used.

a-- *Assumes that a is an integer*

**3. 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. |

***Rules:***

1. Relational Operators should **NOT** be next to each other.
2. Arithmetic Operators should be in the middle of a variable, value or arithmetic operations.

**4. 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 should NOT be next to each other.
2. Logical Operators can only be used in Conditional and Looping statements.

**5. 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 |
| \*= | 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 |

***Rules:***

1. Assignment Operators should **NOT** be next to each other.
2. Assignment Operators should be in the middle of a variable, value or arithmetic operations.
3. **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. **Statements**
2. **Assignment Statements** - In computer programming, an assignment statement 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.

*<id>* = *<value>*

*<id>* += *<expressions>*

*<id>* -= *<value>*

***Syntax***

***Rules:***

1. The identifier that will have the new value should be on the left side.
2. If the data types of both id and expressions are the same, the new value will be saved on the variable. If not, the result will be converted to the data type of the variable.
3. Once the assignment statement takes place, the original value of the variable will disappear.
4. Always end with period(.).
5. **Input and Output Statements** - In computer programming, an assignment statement 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.

**Write statement**is used to display characters on the screen

Write *<statement>.*

***Syntax***

***Rules:***

1. The keyword Write must only followed by a space.
2. In displaying a value of a variable, the space should follow by the name of the variable.
3. Use comma (,) to display multiple variables in one statement.
4. In displaying a word, the space should follow by the word that will be printed out, enclosed by a double quote (“”).
5. To use a double quote to print out, use the backlash symbol (\) followed by the double quote (“).
6. End with a period(.)

Write a.

Write a,b,c,d.

Write “Hello World”.

Write “\” Hello World \””.

***Sample***

**Read statement** is used to display characters on the screen

Read *<variable>.*

***Syntax***

***Rules:***

1. The keyword Read must only followed by a space. Followed by the name of the variable that will serves as the placeholder of the user’s input.
2. Value of identifiers, array’s element, and an object of File are the valid placeholders for the user’s input.
3. One Read statement per input.
4. End with a period(.)

Read a.

Read array[0].

Read Artist@Movies.

***Sample***

1. **Conditional Statements** - support decisions based on a certain condition: if the condition is met, or "true," a certain piece of code is executed.

**If Statements ( If, ElseIf, Else)**

If (<boolean\_expression>)

Start

<statements>.

End.

ElseIf ((<boolean\_expression>)OR(<boolean\_expression>))

<statement>.

ElseIf ((<boolean\_expression>)AND(<boolean\_expression>))

<statement>.

Else

Start

<statements>.

End.

***Syntax***

***Rules:***

1. There must always be a boolean expression inside the parenthesis after the keyword If.
2. Nested conditional statements are possible.
3. If there is more than one statement inside the conditional statements, the keyword Start and End are required. Otherwise, it’s not.
4. It should have an If statement first before having ElseIf and Else statements.
5. If there is more than one boolean expression, make sure to use the logical operators.
6. No Else statement/condition is possible.

If (a==5)

Start

Write “a is equal to 5”.

a++.

End.

ElseIf ((a<4)OR(<a==4>))

Write “a is less than 5”.

ElseIf ((a>5)AND(a<50))

Write “a is greater than 5 but less than 50”.

Else

Start

Write “a is greater than 50”.

a--.

End.

***Sample***

**Switch*-Choice Statements***

Switch(<identifier>)

Start

Choice(<value>): <statement>. Stop.

Choice(<value>): <statement>. Stop.

Choice(<value>): <statement>. Stop.

Default: <statement>. Stop.

End.

***Syntax***

***Rules:***

1. Switch statements should always start with the keyword Switch followed by the identifier to test, enclosed by parenthesis (()).
2. Switch statement must have the keywords Start and End to indicate the body of the Switch.
3. Choice must be followed by a number, decimal, character or a string. Followed by a colon ( : ).
4. Statements should be written after the colon.
5. It is recommended to have the keyword Stop on every Choice.
6. Switch statements also has the Default statement, in case that all the conditions failed to meet the Switch.
7. The Default statement must only followed by colon ( : ) and then it’s statements.

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.

***Sample***

1. **Looping Statements** - Repeats a statement or group of statements while a given condition is true.

***For statement***

For(<initialization>,<boolean\_expression>,<iteration>)

START

<statements>.

………

END.

***Syntax***

***Rules:***

1. Must begin with the keyword For.
2. The keyword For must followed by an open parenthesis. Inside the parenthesis, it should have initialization, boolean expression and iteration, all separated by a comma (,). Then close it with a close parenthesis.
3. For statement must have the keywords Start and End to indicate the body of the For.
4. Nested looping statements are allowed.

For(number a=0,a<10,a++)

START

Write a.

Newline.

END.

***Sample***

***Do-While statement***

Do

START

<statements>.

……

END.

While (<boolean\_expression>).

***Syntax***

***Rules:***

1. Must begin with the keyword Do.
2. Do statement must have the keywords Start and End to indicate the body of the Do.
3. After the keyword end. It must have the While statement to indicate how many loops the statement will do.
4. Nested looping statements are allowed.

Do

Start

Write a.

Newline.

a++.

End.

While(a<10).

***Sample***

***While statement***

While(<boolean\_expression>)

START

<statements>.

……

END.

***Syntax***

***Rules:***

1. Must begin with the keyword While.
2. Followed by the boolean expression.
3. Do statement must have the keywords Start and End to indicate the body of the Do.
4. Nested looping statements are allowed.

While(a<10)

START

Write a.

Newline.

a++.

END.

***Sample***