**SPECIFIC RULES**

***Identifiers***

***I. Rules in naming Identifiers***

1. It must start with an upper case letter.
2. An Identifier can only have alphanumeric characters and underscores.
3. Maximum of 12 characters per identifier.
4. Identifiers are case sensitive.
5. Reserved Words are not allowed to be used as Identifiers.
6. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

***I.1 Example***

|  |  |
| --- | --- |
| **Valid Identifier** | **Invalid Identifier** |
| Num1 | num1 |
| Product\_ | product 7 |
| Stud\_no | 123Let |
| Sum\_3 | Student; |
| Nh4m3 | nH4m3 |

***Data Types and Literals***

In LexiCom programming language, data types refer to a set of [data](http://searchdatamanagement.techtarget.com/definition/data) with values having predefined characteristics.

|  |  |  |
| --- | --- | --- |
| **Data Types** | | |
| Int | Int | A data type that contains whole numbers. |
| double | Double | Data types that contains whole numbers and a decimal part. (64-bit) |
| char | Char | A data type that deals with characters. |
| String | String | A data type that represent a series of characters. |
| Bool | Boolean | A data type can only handle two values: TRUE or FALSE. |

Here are the *most important things* to know about *variables and constants* in *LexiCom*:

***I. VARIABLES AND CON/ STANTS***

***Declaration of Variables***

1. All variables in *LexiCom* starts with a capital letter.
2. Declaration of variables can be declared globally or locally.
3. Declaration must begin with the keywords ***Var*** for variable followed by the data type, followed by the identifier.
4. All declarations must be terminated by a *period* “**.**”.
5. 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 *semicolon* “**;**”.
6. 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. Variables are assigned with the word ***is***.
  4. Initialization is done by adding a keyword ***is*** after the declaration of the identifier/ variable followed by its initial value.
  5. All initializations must be terminated by a period.
  6. 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 semicolon.
  7. 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.
  8. The value of an initialized variable must be in respect to its data type.

***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. Variables are assigned with the word ***is***.
  4. Initialization is done by adding a keyword ***is*** after the declaration of the identifier/ 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 *semicolon* “**;**”.
  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.

***I.1 Specific Examples***

1. To declare or initialize a constant variable, use the ***Let*** keyword following the definition:

|  |
| --- |
| ***Syntax:***  ***Let <Data Type><id>[is<value>][;<id>[ is<value]…]*.** |

|  |
| --- |
| ***Example:***   * 1. ***Declaration:* Let Int Num.**   2. ***Initialization:* Let Double Pi is 3.14.**   3. ***Multiple:* Let Int Num1; Num2 is 0; Num3 is 1.** |

1. To declare or initialize a variable, use the ***Var*** keyword following the definition:

|  |
| --- |
| ***Syntax:***  ***Var<Data Type><id> [is<value>][;<id>[ is<value]…].*** |

|  |
| --- |
| ***Example:***   * 1. ***Declaration:* Var Int Num.**   2. ***Initialization:* Var Int Num is 0.**   3. ***Declaration and Initialization:* Var Int Num1; Num2 is 0.**   4. ***Multiple:* Let Int Num1; Num2 is 0; Num3 is 1; Num4 is 5.** |

1. ***Int- A data type that contains whole numbers.***

*Rules for Int Data Type*

* 1. The data type *Int* can be used in declaring an integer type variable.
  2. When initializing, the word ***is*** followed by its initial value must be appended after the identifier.
  3. Accepts whole numbers only.
  4. Input of the data type should be a *Integer* literal.

|  |  |
| --- | --- |
| **Valid Int Data Type** | **Invalid Int Data Type** |
| Var Int Num. | Var Int Num1 is “Number 1”. |
| Var Int Product\_ is 7. | Var int Product\_ is 7. |
| Var Int Stud\_No is 201320157. | Var Int 123Let. |
| Var Int Sum. | Var Int Student; is 4. |
| Var Int Age is 19. | Var Int Total is 4.5. |

*Rules for Integer Literals*

1. Value of an *Integer* literal can range from ***-999999999999999999*** to ***999999999999999999***. Exceeded to this range is not valid.
2. Leading zeros will be disregarded. *Integer* will be valid but the fractional part will be omitted.

|  |  |
| --- | --- |
| **Valid Int Literal** | **Invalid Int Literal** |
| 123 | “Aut0mata” |
| 0 | ‘A’ |
| -123 | Yes |

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

*Rules for Double Data Type*

1. The data type *Double* can be used in declaring a floating type variable.
2. When initializing, the word ***is*** 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 *Double* literal.

|  |  |
| --- | --- |
| **Valid Double Data Type** | **Invalid Double Data Type** |
| Var Double Num. | Var Double Num1 is “Number 1”. |
| Var Double Price\_ is 7.2. | Var double Quantity\_ is 7. |
| Var Double Stud\_No is 2013.20157. | Var Double 123Let. |
| Var Double Product. | Var Double Student is 4. |

*Rules for Double Literals*

1. Value of a *Double* literal can range from ***-999999999999999999.9999*** to ***999999999999999999.9999***. Exceeded to this range is not valid.
2. Leading zeros will be disregarded.

|  |  |
| --- | --- |
| **Valid Double Literal** | **Invalid Double Literal** |
| 123.45 | “Aut0mata” |
| 123.4567 | 123 |
| 123.450 | ‘A’ |
| -123.4 | Yes |

**C.** ***Char- A data type that deals with characters.***

*Rules for Char Data Type*

* + 1. The data type *Char* can be used in declaring a character type variable.

1. When initializing, the word ***is*** followed by its initial value must be appended after the identifier.
2. Input of the data type should be a *Char* literal.
3. In initializing its value, the character must be enclosed in a single quote “ **’ ’** ”.

|  |  |
| --- | --- |
| **Valid Char Data Type** | **Invalid Char Data Type** |
| Var Char Gender. | Var Char Num1 is “Number 1”. |
| Var Char CivilStat\_ is ‘S’. | Var char Quantity\_ is 7. |
| Var Char Civil\_Status is ’M’. | Var Char 123Let. |
| Var Char Day is ‘W’. | Var Char Standing; is ‘GS’. |

*Rules for Char Literals*

* 1. Value of a *Char* literal can be any Letter or Number. Otherwise, it is not valid.
  2. *Char* literals should be enclose by a single quote “ **‘ ’** ”.
  3. *\’*  is use in initializing a single quote inside a char literal.

|  |  |
| --- | --- |
| **Valid Char Literal** | **Invalid Char Literal** |
| ‘F’ | “Aut0mata” |
| ‘f’ | 123.45678 |
| ‘\’’ | Yes |

**D.** ***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 word ***is*** followed by its initial value must be appended after the identifier.
  1. Input of the data type should be a *String* literal.
  2. In initializing its value, the character must be enclosed in a double quote “**“ ”**”.

|  |  |
| --- | --- |
| **Valid String Data Type** | **Invalid String Data Type** |
| Var String Name. | Var String Num1 is ‘A’. |
| Var String College\_ is “College of Engineering and Technology”. | Var string Department\_ is “Computer Science Department”. |
| Var String Day is “Today is \”Wednesday\””. | Var String Student; is “Juan Dela Cruz”. |

*Rules for String Literal*

1. Value of a *String* literal can be any printable character.
2. *String* literals should be enclosed by a double quote “ **“ ”** ”.
3. *\”*  is use in initializing a double quote inside a string literal.

|  |  |
| --- | --- |
| **Valid String Literal** | **Invalid String Literal** |
| “A” | ‘Aut0mata’ |
| “a” | 123.45678 |
| “1” | ‘A’ |
| “ ” | Yes |
| “My Name is \”Juan Dela Cruz.\”” | ‘S; |

**E.** ***Boolean-* A data type can only handle two values: Yes for true or No for 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: Yes or No.
    3. When initializing, the word ***is*** 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** |
| Var Boolean Decision. | Var Boolean Decision is no. |
| Var Boolean Decision\_ is Yes. | Var boolean Decision\_ is Yes. |
| Var Boolean Smart\_Choice is No. | Var Boolean 123Let. |
| Var Boolean Choice is Yes. | Var Boolean Standing; is NO. |

*Rules for Boolean Literals*

* 1. Value of a *Boolean* literal can only be a *Yes* or a *No*. Otherwise, it is not valid.

|  |  |
| --- | --- |
| **Valid Boolean Literal** | **Invalid Boolean Literal** |
| Yes | YES |
| No | no |

***Array***

***I. Rules for Array Declaration***

1. Declaration of arrays can be declared globally or locally.
2. Declaration must begin with the keyword ***Array***, followed by its data type and identifier then keyword ***of*** then the size of the array.
3. Initialization can be less than the size of the array but cannot be greater than the size of the array.
4. All declarations and initializations must be terminated by a period.
5. Multiple declaration of array is not allowed.
6. The value of an initialized variable must be in respect to its data type.
7. Array indices are fixed numeric values as declared in its declaration.
8. Array contents must strictly contain a data of its own data type.
9. All array indices are treated as positive whole numbers only, if negative number is declared it will be considered as a positive number.
10. The array size must be declared.
11. The maximum dimension of an array is two. The minimum size of an array is one and the maximum size of an array is 50.
12. To declare a multidimensional array, add ***by*** right after the first size of the array.
13. Initialization to a multi-dimensional array is not possible.

|  |
| --- |
| ***Specific example:***   1. **Array Int Nums of 3.** 2. **Array Int Nums of 3 by 2.** |

***Tasks***

***I. Rules in Task Declaration***

1. Declaration of functions can only be declared globally.
2. Every program must contain at least one task, the ***Lead*** task. It is the first task executed when the program starts.
3. To declare the task Lead, use the keyword ***Lead*** followed by a semicolon, enclosed with ***Start*** and ***End*** then ended with a period.
4. Other Tasks must be declared before the ***Lead*** task.
5. In declaring task, use the keyword ***Task*** followed by its response-type then its identifier. Use period as terminator.
6. In declaring task, response-type must be specified. Use the keyword ***Null*** if there is no need to response anything.
7. In declaring task with a parameter, declare a variable enclosed by parenthesis after the task name. Use semicolon as separator if multiple parameter is needed.
8. Tasks are called through its function name followed by a pair of parenthesis. If there is/are parameter/s needed, it must be indicated and separated by a semicolon inside the parenthesis.
9. A task can call itself (Recursion).
10. Defining the statements for task is done after the Lead task.
11. In defining the statements for task, same rules apply as to declaring one except for the terminator. Instead of a period, use colon then followed by the keyword ***Start*** and **End***,* between the Start and End is/are the statement/s defined. Lastly, use period to terminate the task.
12. Use the keyword ***Response*** followed by a response value if response is necessary. Response value must be in respect to the response type of the task.
13. A task must have at least one statement.

***I.1 Specific Examples***

1. To declare a task, use the ***Task*** keyword following the definition:

|  |
| --- |
| ***Syntax:***  ***Lead:*** *(For Lead task)* ***Task<Data Type><task name> [( Var<Data type><Variable name>[; Var<Data type><Variable name>…] )].*** *(*For other Tasks*)* |

|  |
| --- |
| ***Example:***   * 1. **Lead: Start End.**   2. **Task Null Clean.**   3. **Task Int Sum (Var Int X; Var Int Y).** |

To define the statement of a task, use the following definition:

|  |
| --- |
| ***Syntax:***  ***Task <Data type><task name> [( Var <Data type><variable name> [; Var<Data type><variable name>…] )]: Start Statement\_1. [Statement\_2].  …  [Response <response value>]. End.*** |

|  |
| --- |
| ***Example:***   1. **Task Sum (Var Int X; VarInt Y): Start Var Int S.  S = X + Y.  Response S. End.** 2. **Task Null Disp: Start  Say(“Hello Human”). End.** |

***Operators***

An operator is a symbol which operates on a value, constant, variable, and array. LexiCom is rich in built-in operators and provides the following types of operators –

* + 1. Arithmetic Operators
    2. Unary Operators
    3. Logical Operators
    4. Relational Operators
    5. Assignment Operators

***1. Arithmetic Operators -*** used for mathematical operations.

|  |  |  |
| --- | --- | --- |
| Operator | Operator’s name | Description |
| + | Binary addition | adds two operands |
| - | Binary subtraction | subtract second operands from first |
| \* | Binary Multiplication | multiply two operand |
| / | Binary Division | divide numerator by denominator |
| % | Modulus | remainder of division |

***Rules:***

* 1. The symbol “-” can be placed before a value if the value is negative, after the arithmetic operator and must be followed by a value.
  2. No two arithmetic operators must be next to each other.
  3. Arithmetic symbol must be in the middle of a variable, value or arithmetic expression.

1. ***Unary Operators -*** symbols used to operate on a single operand.

|  |  |  |
| --- | --- | --- |
| Operator | Operator’s name | Description |
| ++ | Increment | Increases the value of its operand by 1. |
| -- | Decrement | Decreases the value of its operand by 1. |

***Rules:***

1. Increment and decrement operator works only in variables with integer value.
2. Increment and decrement operator can be before or after a variable. It depends on how it will be use.

*Increment:*

**Let Int X is 5.**

* 1. Post-increment– first, assign the value of the identifier, then the increment will be performed.

X ++

* 1. Pre-increment –first, perform the increment then assign the value of the identifier.

++ X

*Decrement:*

**Let Int X is 5.**

* + 1. Post-decrement – first, assign the value of the identifier then the decrement will be performed.

X --

* + 1. Pre-decrement – first, perform the decrement then assign the value to the identifier.

-- X

***3. Logical Operators -*** used to get BOOLEAN values.

|  |  |  |
| --- | --- | --- |
| Operator | Operator’s name | Description |
| && | Logical AND Operator | If both operands are true, then the condition becomes true. |
| || | Logical OR Operator | If any of the two operands is true, then the condition becomes true. |
| ! | Logical NOT Operator | It is used to reverse the state of the operand. |

***Rules:***

1. No two logical operators must be next to each other.
2. Logical operators can only be use in conditional statements.
3. In order to get a Boolean value, operands should have a Boolean value, also.

***4. Relational Operators***

These are symbols used to test or define relation between entities.

|  |  |  |
| --- | --- | --- |
| Operator | Operator’s Name | Description |
| == | Equal to | Check if the values of the two operands are equal. |
| != | Not equal to | Check if the values of two operands are not equal. |
| > | Greater than | Check if the value of the left operand is greater than the value of right operand. |
| < | Less than | Check if the value of the left operand is less than the value of the right operand. |
| >= | Greater than or equal | Check if the value of the left operand is greater than or equal to the value of the right operand. |
| <= | Less than or equal | Check if the left operand is less than or equal to the value of the right operand. |

***Rules:***

* 1. No two relational operators must be next to each other.
  2. Relational Operator must be in the middle of values, variables or relational expression.

***5. Assignment Operators***

These are symbols used to assign values from the operands.

|  |  |  |
| --- | --- | --- |
| Operator | Operator’s name | Description |
| = | Simple assignment operator | Assigns value from the right side operands to the left side operand. |

***Operator Precedence***

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator | Description | Associativity |
| 1 | @  [ ]  ( )  ++  -- | Member selection (object or pointer)  Array subscript  Function call  Postfix increment  Postfix decrement | Left-to-right |
| 2 | ++  --  !  ~ | Prefix increment  Prefix decrement  Logical not  Unary negation | Right-to-left |
| 3 | @ | Pointer-to-member (objects or pointer) | Left-to-right |
| 4 | \*  /  % | Multiplication  Division  Modulus | Left-to-right |
| 5 | +  - | Addition  Subtraction | Left-to-right |
| 6 | <  >  <=  >= | Less than  Greater than  Less than or equal  Greater than or equal | Left-to-right |
| 7 | ==  != | Equality  Inequality | Left-to-right |
| 8 | && | Logical AND | Left-to-right |
| 9 | || | Logical OR | Left-to-right |
| 10 | = | Assignment | Left-to-right |
| 11 | ; | Semi-colon | Left-to-right |

***Expression***

In programming, an expression is any legal combination of symbols that represents a value.

1. Every expression consists of at least one operand and can have one or more operators.
2. Operands are values and Operators are symbols that represent particular actions.

|  |  |
| --- | --- |
| **Valid Expression** | **Invalid Expression** |
| A + B | ++ A + B |

***Statements***

***I. Assignment Statements***

***Assignment statements*** are used for carrying out computation and assigning values to variables.

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

Its purpose is saving the result of the *expression* to the right of the *assignment operator* to the *variable* on the left. Here are some rules:

1. The *expression* is evaluated first with the rules discussed in the [*single mode*](http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap02/exp-1.html) or the [*mixed mode*](http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap02/exp-2.html) expressions pages.
2. If the type of the expression is identical to that of the *variable*, the result is saved in the variable.
3. Otherwise, the result is converted to the type of the variable and saved there.
4. If the data type of the variable is *Int* while the data type of the result is *Double*, the fractional part, including the decimal point, is removed making it an integer result.
5. If the data type of the variable is *Double* while the data type of the result is *Int*, then a decimal point is appended to the integer making it a real number.
6. Once the variable receives a new value, the original one disappears and is no more available.

***II. Input and Output Statements***

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

1. The keyword *Read* must only be followed by a space, followed by the identifier that will act as placeholder for the input. The statement should always be terminated by period “.”.
2. Identifiers, element of an array, and element of an object are the only valid placeholders for the values to be input by the user. Other than that like, literals, will not be accepted.
3. There should be only one read statement for each user’s input and putting multiple placeholders inside the read statement would produce an error.

|  |
| --- |
| ***Specific example:***   1. **Read Num.** 2. **Read Nums[0].** 3. **Read Group@People.** |

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

1. The keyword *Say* must only be followed by a space, followed by the value to be printed out, enclosed by a double quote “ **” ”** ”. The statement should always be terminated by period “**.**”.
2. In concatenating values inside the *Say* statement, the symbol “**,**” should be used.
3. To include double quote, use the backslash symbol “**\**”, to be followed by a double quote. Same rule applies if you want to add the backslash symbol.

|  |
| --- |
| ***Specific example:***   * 1. Say “Hello World!”. **OUTPUT: Hello World!**   2. Say “Hello”,”World!”. **OUTPUT: Hello World!**   3. Say “\”Hello World!\””. **OUTPUT: “Hello World!”** |

***III. Conditional Statements***

*If statements (If-Otherwise || If-Or || If-Or-Otherwise)*

|  |
| --- |
| ***Syntax:***  ***If(<boolean expression>[&&<Boolean expression] [||<Boolean expression]] )***  ***<statements>.***  ***….***  ***[[Or (<boolean expression>)***  ***<statements>.***  ***….]***  ***Otherwise***  ***<statements>.***  ***….]***  ***EndIf.*** |

1. There must be always a boolean expression enclosed in parentheses “( )” after the word ***If***. Boolean expression can be either a boolean value or a conditional expression. The arguments in the conditional expression could be an identifier, an element of an array, an element of an object, a task or a literal, with a conditional/relational operator.
2. Nested conditional statements are allowed.
3. No **Otherwise** statement is possible.
4. The word ***Otherwise*** should be after the ***If*** statement and **Or** statement if existing and it should not be followed by a boolean expression.
5. The word ***Or*** should be followed by a boolean expression enclosed in parentheses

“( )” and it is used if the condition in the ***If*** statements are not satisfied.

1. Any number of statements must be enclosed in the keyword ***EndIf*** and ended with a period ”**.**”.

|  |
| --- |
| ***Specific example:***   1. **If (Num == 1)  Num is 2. EndIf.** 2. **If (Num==2)  Say “True”. Otherwise  Say “False”. EndIf.** 3. **If (Num==1)  Say “1”. Or (Num==2)  Say “2”.**   **Otherwise  Say “3”. EndIf.** |

*Option-State Statements*

1. Must begin with the word “***Option***”.
2. The word ***Option*** must be followed by an identifier then the word ***State*** that is enclosed by the words ***Start*** and ***End*** and ended with the terminator “***.***”.
3. ***State*** is followed by a int literal, char literal, or string literal *value* then a colon “**:**”.
4. Next to it are the statements to be executed ended with the word ***Stop*** and terminated with a period “**.**”.
5. It is also allowed to have a ***Default*** statement in case that there is no true condition in the given states or options
6. The word ***Default*** must be followed by a colon “**:**” then the statements to be executed.

|  |
| --- |
| ***Specific example:***   * + 1. **Option Num**   **Start**  **State 1: Say “The number is 1.”. Stop.**  **State 2: Say “The number is 2.”. Stop.**  **State 3: Say “The number is 3.”. Stop.**  **Default: Say “The number is neither of the choices.”. Stop.**  **End.** |

***IV. Looping Statements***

*For Statements*

|  |
| --- |
| ***Syntax:***  ***For(<initialization>;<boolean expression>;<iteration> )***  ***<statements>.***  ***….***  ***Loop.*** |

1. Must begin with the word “***For”***.
2. An opening parenthesis “**(**“ must always follow after the word ***For***.
3. Next to it must have an initialization followed by a semicolon “***;***” then a Boolean expression followed by another semicolon “***;***”, then an iteration followed by a closing parenthesis “**)**”.
4. Any number of statements made must be enclosed by the keyword ***Loop*** and must end with the terminator “***.***”.
5. Nested looping statements are allowed.

|  |
| --- |
| ***Specific example:***   1. **For (i = 1; i >= 5; i++)  Say “Hello”. Loop.** 2. **For(i = 1; i >= 5; i++)  Say “Hello”.**   **For (j = 5; j <= 1; j--) Say “Hello”.  Loop. Loop.** |

*Do Statements*

|  |
| --- |
| ***Syntax:***  ***Do***  ***<statements>.***  ***….***  ***LoopIf(<boolean expression>).*** |

1. Begins with the word ***Do*** then followed by the statements to be executed then closed by the word ***LoopIf*** with the boolean expression enclosed in parentheses “**( )**”.
2. Statements should be terminated with a period “***.***”.

|  |
| --- |
| ***Specific example:***   1. **Do**   **Num++.**  **LoopIf (Num<10).** |

*Until Statements*

|  |
| --- |
| ***Syntax:***  ***Until(<boolean expression>)***  ***<statements>.***  ***….***  ***Loop.*** |

1. Begins with the word ***Until*** followed by the boolean expression enclosed in a parentheses “**( )**”then the statements to be executed, closed by the word ***Loop***.
2. Statements should be terminated with a period “***.***”.

|  |
| --- |
| ***Specific example:***   1. **Until (i<10)**   **Num++.**  **Loop.s** |