luseed

a programming language documentation proposal

- Bandala, Charles Andre
- Belanio, Glaizel Nicole
- Famoso, Nina Grace
- Malapit, Sthanly Paul
- Sulit, Dexter
- Ubungen, Zen Desiree

Table of Contents

Introduction

In the vast landscape of programming languages, a new seed has been planted, destined to grow into a language that embodies clarity and simplicity—meet luseed. Derived from the fusion of the words "lucid" and "seed," luseed is a static, object-oriented programming language that aspires to be the nurturing ground for budding programmers, providing a simple, straightforward, and easily comprehensible foundation for their exploration into the world of programming.

Just as the word lucidity denotes the quality of being clear and easily understood, luseed seeks to provide a clearer understanding of the complexities often associated with programming languages. Its design was from the belief that the path to mastery should begin with a language that nurtures comprehension rather than overwhelming beginners.

The metaphor of a seed is intentional: it symbolizes not only the formation of a new programming language but also the potential for growth and development. Luseed aims to be the fertile ground from which programming skills sprout and flourish, offering a gentle introduction to the world of programming for those people who are taking their first steps in the field.

Designed with a focus on the education sector, luseed is specially made to be one of the possible first programming languages that can be used by beginners. The syntax of luseed is intentionally made to be as humanly and as simple as possible, while incorporating the essential syntactical elements and basic principles found in other programming languages such as C#, Python, and Lua. This approach ensures that learners not only grasp the fundamental coding concepts but also aid them in gaining a strong foundation that can help them in seamlessly transitioning into other languages in the future.

Syntactic Elements of Language

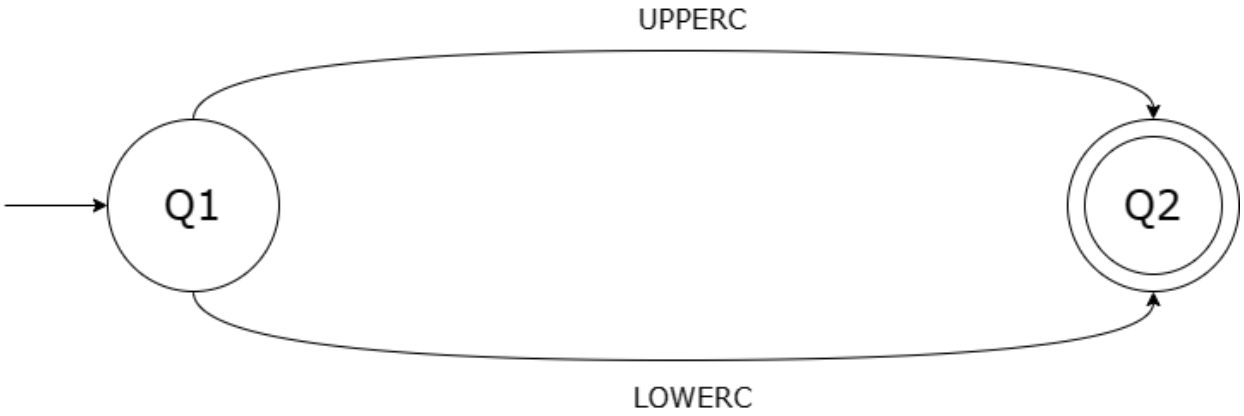
The following pages contain information about the Character Set, Identifiers, Operation Symbols, Keywords and Reserved Words, Noise Words, Comments, Blanks, Delimiters and Brackets, Field Format, Expressions, and Statements of the luseed programming language.

I. CHARACTER SET

This segment covers all the characters accepted in the luseed programming language.

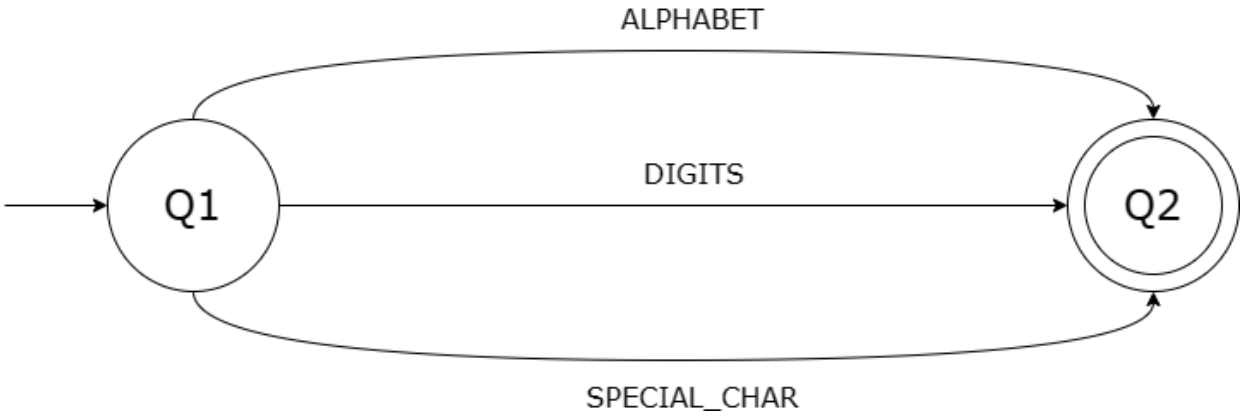
- CHARACTERSET = {ALPHABET, DIGITS, SPECIAL_CHAR}
- ALPHABET = {UPPERC, LOWERC}
- UPPERC = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
- LOWERC = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
- DIGITS = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- INTEGERS = DIGITS*|-DIGITS*
- SPECIAL_CHAR = {., +, -, *, /, %, <, >, =, \, ", ', ,, ;, |, !, (,), [,], {, }, _, ^, ~, &, :, ?, ^, #, @, -, `, <space>}

STATE MACHINE FOR ALPHABET



Regular Expression: UPPERC + LOWERC

STATE MACHINE FOR CHARACTERSET



Regular Expression: ALPHABET + DIGITS + SPECIAL_CHAR

II. IDENTIFIERS

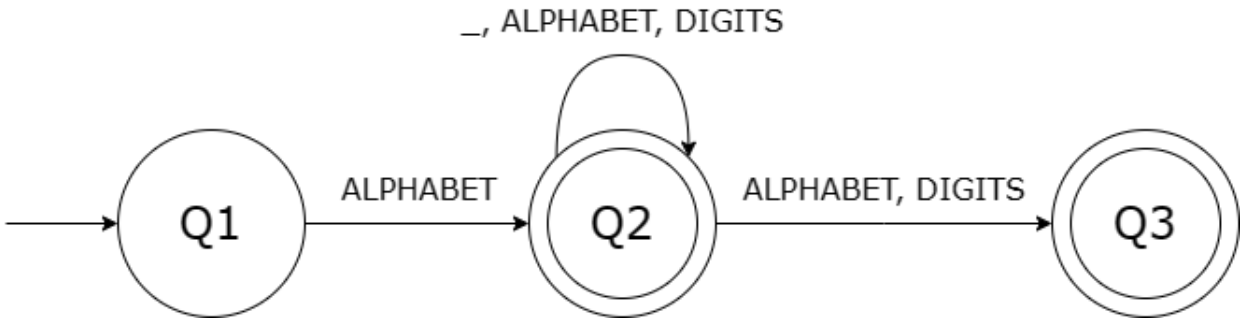
This part of the documentation holds everything that is essential in terms of the identifiers, which is the accepted formats in naming variables in the luseed programming language.

Rules:

- An identifier should start with an alphabetical character, it could be either an uppercase (A-Z) or lowercase (a-z).
- The identifiers must be **case sensitive**. This means that `FirstVar` and `firstVar` are read differently.
- Numerical characters (0-9) are allowed, given that the identifier starts with an alphabet character.
- Underscore is the only special character that is accepted as long as it is not placed on the beginning nor the end of the identifier.
- Only a maximum number of 100 characters are allowed to be used as an identifier.

VALID	INVALID
`num1`	`9numVar`
`NumberVariable`	`_$number`
`nAme_VaRiable`	`+(=umberVar`
`PersonName`	`_integerValue`
`gravityVariable`	`*/stringValue__`

STATE MACHINE FOR IDENTIFIERS



Regular Expression: ALPHABET(_ + ALPHABET + DIGITS)$^{+}$(ALPHABET + DIGITS)

III. OPERATION SYMBOLS

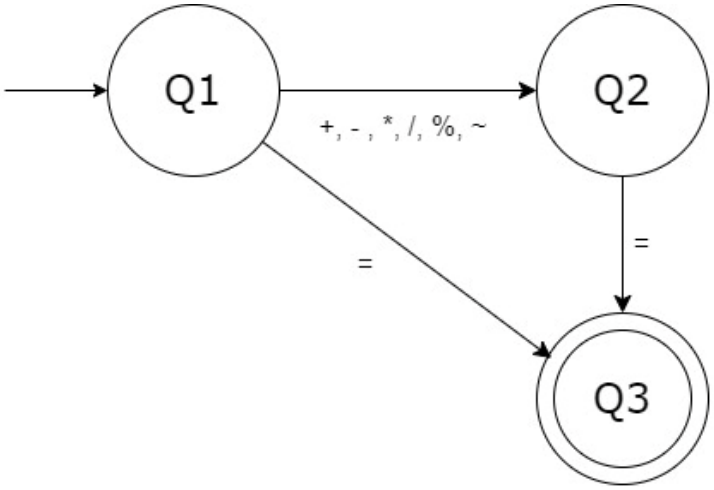
This segment of the documentation covers the operational symbols that the luseed programming language holds. This also contains an example expression and a short description for understandability.

Assignment Operators

- OP_ASSIGNMENT = {=, +=, -=, *=, /=, %=, ~=}

OP_ASSIGNMENT	Example Expression	Description
= (Assignment Operator)	x = 3	Assigns 3 as the value of variable x.
+= (Addition Assignment Operator)	x += 3	Adds 3 to the current value of x and assigns the sum to variable x.
-= (Subtraction Assignment Operator)	x -= 3	Subtracts 3 to the current value of x and assigns the difference to variable x.
*= (Multiplication Assignment Operator)	x *= 3	Multiplies the current value of x by 3 and assigns the product to variable x.
/= (Division Assignment Operator)	x /= 3	Divides the current value of x by 3 and assigns the quotient as the value of variable x.
%= (Modulo Assignment Operator)	x %= 3	Divides the current value of x by 3 and assigns the remainder to variable x.
~= (Floor Division Assignment Operator)	x ~= 3	Divides the integer value of x to 3 and assigns the rounded down quotient to variable x.

STATE MACHINE FOR OP_ASSIGNMENT



Regular Expression: = + (+ + - + * + / + % + ~)=

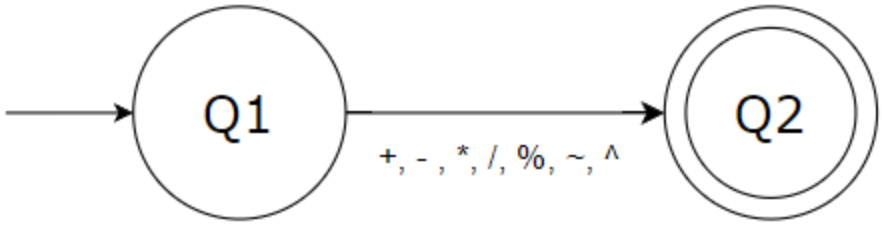
Arithmetic Operators

- OP_ARITHMETIC = {+, -, *, /, %, ~, **}

OP_ARITHMETIC	Example Expression	Description
+ (Addition Operator)	x + y	Adds the value of x and y.
- (Subtraction Operator)	x - y	Subtracts the value of x and y.
* (Multiplication Operator)	x * y	Multiplies the value of x and y.
/ (Division Operator)	x / y	Divides the value of x and y.
% (Modulo Operator)	x % y	Divides the value of x and y and returns the remainder.
~ (Floor Division Operator)	x ~ y	Divides the integer value of x and y and the result will be rounded down to the nearest whole number.

** (Exponent Operator)	x ** n	Calculates x raised to the power of n.

STATE MACHINE FOR OP_ARITHMETIC



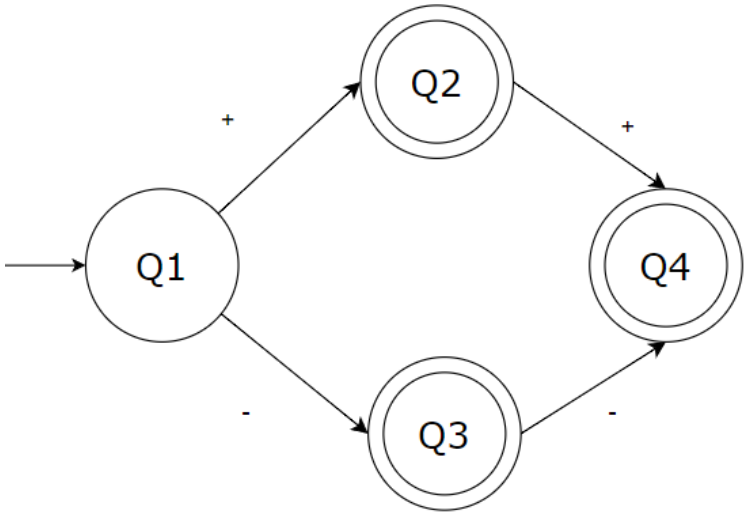
Regular Expression: = + + - + * + / + % + ~ + **

Unary Operators

- OP_UNARY = {+, -, ++, --}

OP_UNARY	Example Expression	Description
+ (Unary Plus Operator)	+x	Indicates that the value of x is positive.
- (Unary Minus Operator)	-x	Indicates that the value of x is negative.
++ (Increment Operator)	++x or x++	Adds 1 to the value of x.
-- (Decrement Operator)	--x or x–	Subtracts 1 to the value of x.

STATE MACHINE FOR OP_UNARY



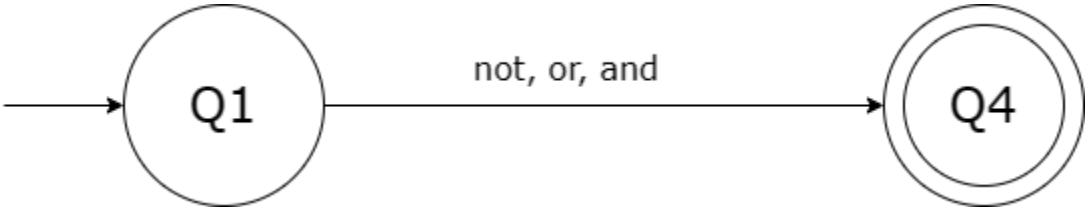
Regular Expression: (+ + (++)) + (- + (--))

Logical Operators

- OP_LOGIC = {not, or, and}

OP_LOGIC	Example Expression	Description
not (Logical NOT Operator)	[x = true;] not(x)	Returns the opposite value of x. [not(x) returns false]
or (Logical OR Operator)	[x = 3;] (x>5 or x<5)	Returns the value true if at least one condition is satisfied, or false if all conditions are false. [(x>5 or x<5) returns true]
and (Logical AND Operator)	[x = 3;] (x>1 and x<5)	Returns the value true if all conditions are satisfied, or false if at least one condition is false. [(x>1 and x<5) returns true]

STATE MACHINE FOR OP_LOGIC



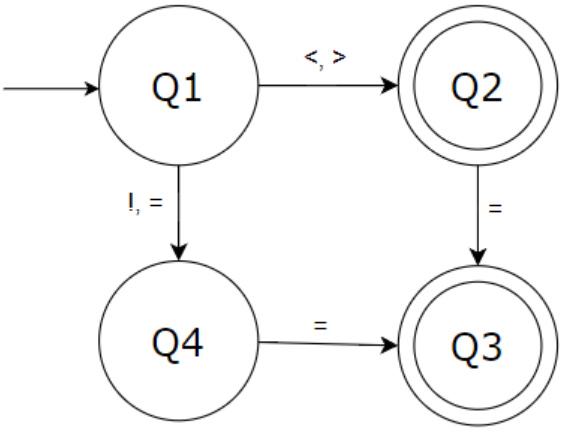
Regular Expression: not + or + and

Relational Operators

- OP_RELATION = {==, !=, >, <, <=, >=}

OP_RELATION	Example Expression	Description
== (Equal to Operator)	[x = 3, y = 3;] (x == y)	Compares the value of x and y and returns true if they are equal, or false if they are not. [(x == y) returns true]
!= (Not Equal To Operator)	[x = 3, y = 3;] (x != y)	Compares the value of x and y and returns false if they are equal, or true if they are equal. [(x == y) returns false]
> (Greater Than Operator)	[x = 3, y = 9;] (x > y)	Compares the value of x and y and returns true if the value of x is greater than the value of y, or false if it is not. [(x > y) returns false]
< (Less Than Operator)	[x = 3, y = 9;] (x < y)	Compares the value of x and y and returns true if the value of x is less than the value of y, or false if it is not. [(x > y) returns true]
>= Greater Than or Equal to Operator)	[x = 9, y = 9;] (x >= y)	Compares the value of x and y and returns true if the value of x is greater than or equal to the value of y, or false if neither of the two are met. [(x >= y) returns true]
<= (Less than or Equal to Operator)	[x = 9, y = 3;] (x >= y)	Compares the value of x and y and returns true if the value of x is less than or equal to the value of y, or false if neither of the two are met. [(x <= y) returns true]

STATE MACHINE FOR OP_RELATION



Regular Expression: ((< + >) + (< + >) =) + (! + =) =

IV. KEYWORDS AND RESERVED WORDS

The following table contains the keywords and reserved words of the luseed programming language:

Note: All the keywords in luseed are reserved words they cannot be used as an identifier.

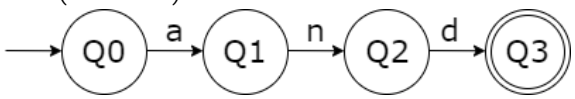
KEYWORDS	Definition
and	Used to perform a logical AND.
ask	Used to request user input.
bool	Used to store values of true or false.
break	Used control statement in breaking out of loops.
catch	Used to handle errors without halting the execution process of the code.
char	Used to store single character/letter/number, or ASCII values.
check	Used for checking whether a method/property exists within a class.
class	Used in creating objects.
const	A variable that is unchangeable.
continue	Sends control back outside a loop.
display	Used to display character, string, float, integer, octal, and double onto the output screen.
do	Executes the statement/s given in a do-until loop before checking the condition
double	Used to Store fractional numbers, containing double-precision floating-point numbers.
elif	Used to specify a new condition to test, if the first condition is false.
else	Used to specify a block of code to be executed, if the same condition is false.
false	A Boolean literal.

finally	Used to execute given statements after the try..catch statement regardless of the outcome.
float	Used to store fractional numbers containing single-precision floating-point number.
for	Used to start a loop that iterates over a defined range.
foreach	Used to start a loop that is used exclusively to loop through elements in an list.
func	Used to define a function.
if	Used to specify a block of code to be executed whether a condition is true or false.
info	Used to display information about a keywords or statements.
init	Used in a reserved function of luseed. Used for the initialization function of a given class that executes when a class is initialized.
inheritall	The equivalent of super in luseed. This lets a child class inherit all the functionality of a parent class
in	Used to iterate through a sequence in a foreach loop
int	Used to store whole numbers, without decimals.
list	Used to declare a list which stores multiple values of data.
main	Used as a reserved function of lused. Used to denote the main function in a luseed program.
not	Used to perform logical negation.
null	Used to denote that there is no value.
or	Used to perform a logical OR.
obj	Data type for an object.
private	An access modifier where it is only accessible only by the code which in the same class of structure.
protected	An access modifier that are only accessible to a class derived from it.

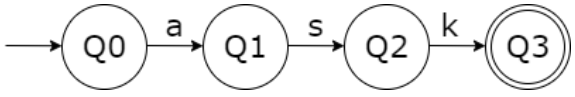
public	An access modifier where it is easily accessible from any part of the program.
quit	Used to quit the info utility in luseed.
raise	Used for raising exceptions in a program.
repeat	Used to create a loop that executes a given statement/s depending on the integer value inputted by the user.
return	Used to return a value to a function.
str	Used to take string input.
then	Used to introduce a block of code that should be executed if the condition specified in the preceding statement is true.
this	Refers to the current instance of an object.
true	Boolean literal.
try	Used in error handling. Used in trying to run a specific code to check whether it returns an error or not.
until	Used to create a loop executed repeatedly until a specified condition becomes true.
while	Used to create a loop that runs while the given condition is true.

STATE MACHINE FOR KEYWORDS

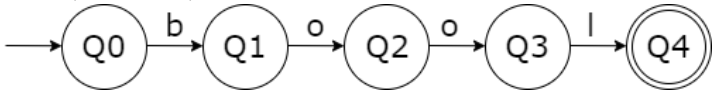
- and (RE: and)



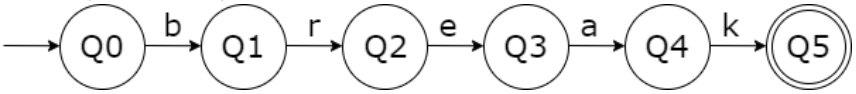
- ask (RE: ask)



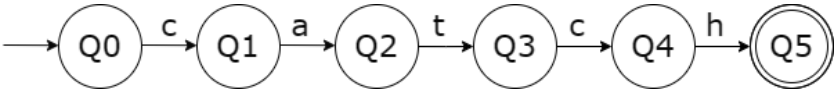
- bool (RE: bool)



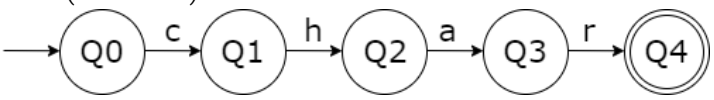
- **break** (RE: break)



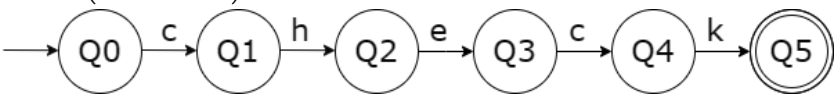
- **catch** (RE: catch)



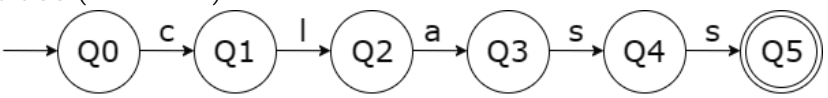
- **char** (RE: char)



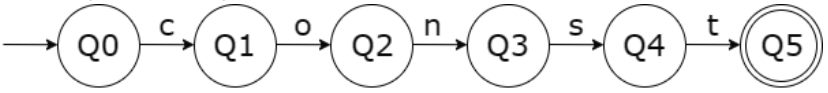
- **check** (RE: check)



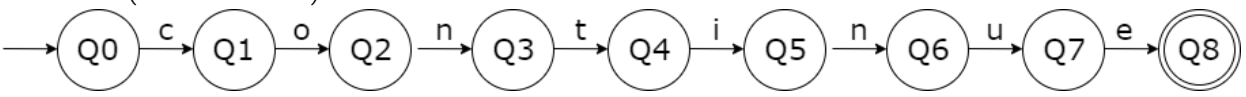
- **class** (RE: class)



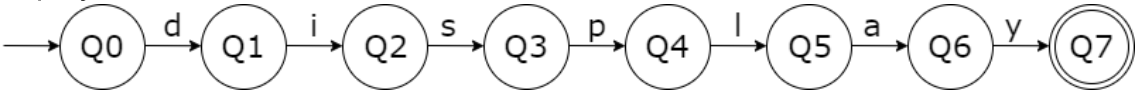
- **const** (RE: const)



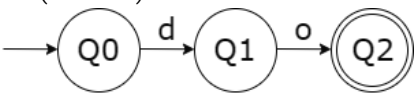
- **continue** (RE: continue)



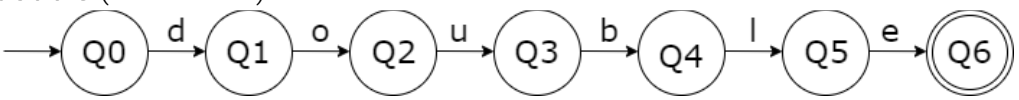
- **display** (RE: display)



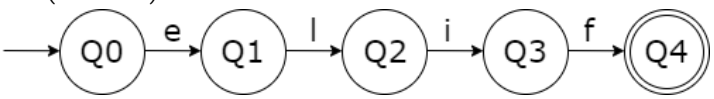
- **do** (RE: do)



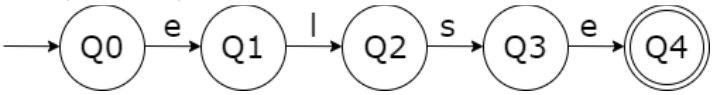
- **double** (RE: double)



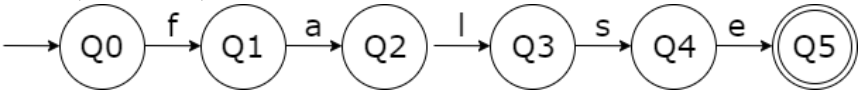
- elif (RE: elif)



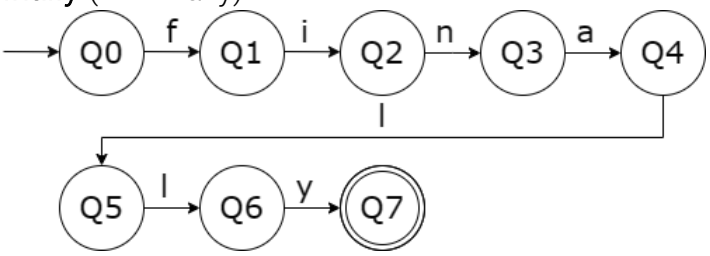
- else (RE: else)



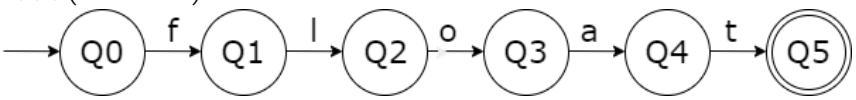
- false (RE: false)



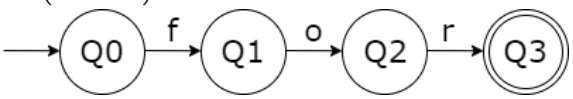
- finally (RE: finally)



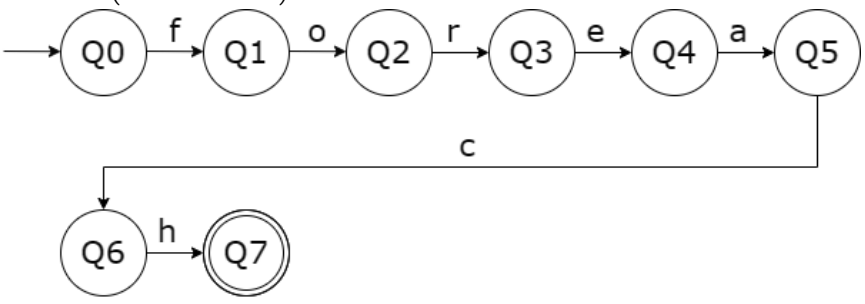
- float (RE: float)



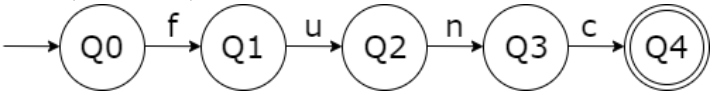
- for (RE: for)



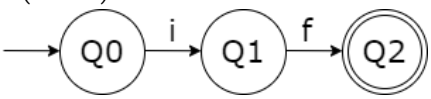
- foreach (RE: foreach)



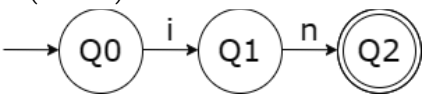
- func (RE: func)



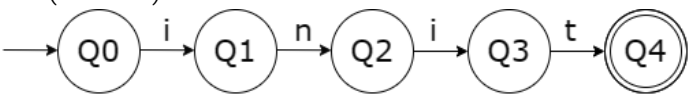
- if (RE: if)



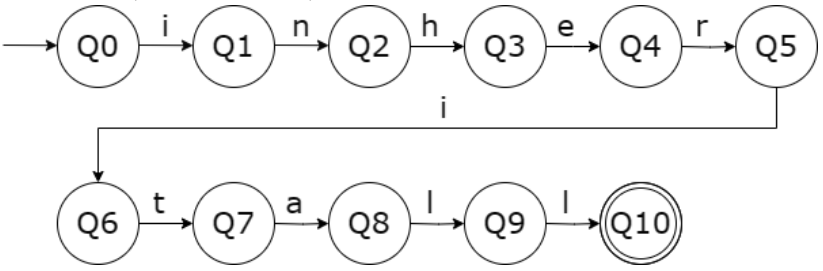
- in (RE: in)



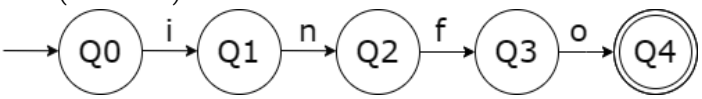
- init (RE: init)



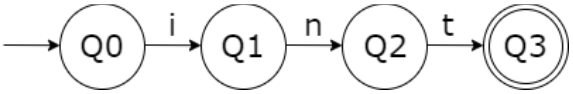
- inheritall (RE: inheritall)



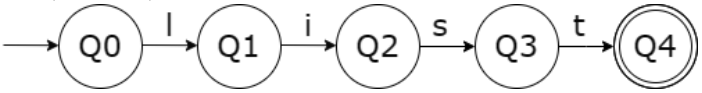
- info (RE: info)



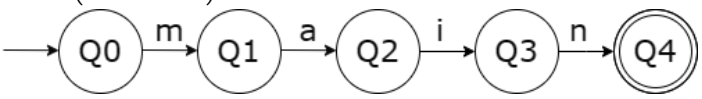
- int (RE: int)



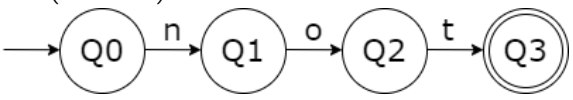
- list (RE: list)



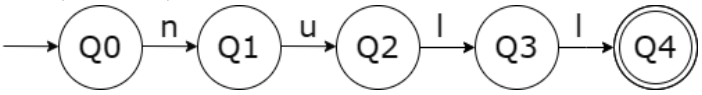
- main (RE: main)



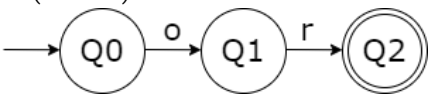
- not (RE: not)



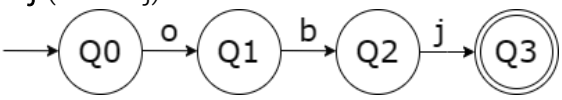
- null (RE: null)



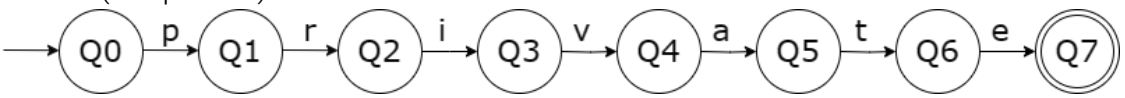
- or (RE: or)



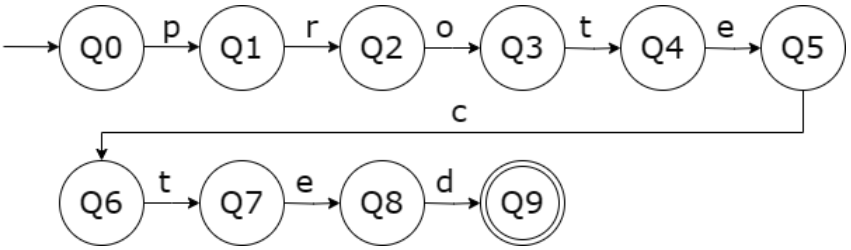
- obj (RE: obj)



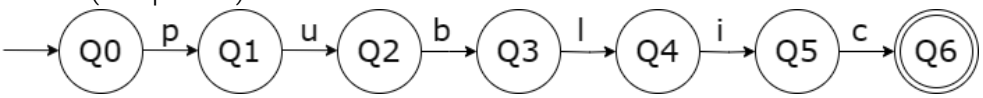
- private (RE: private)



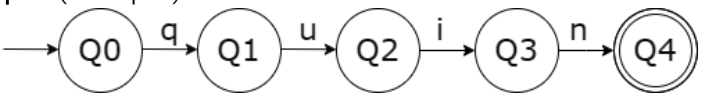
- protected (RE: protected)



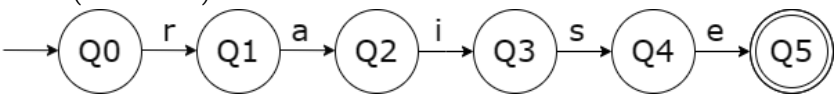
- public (RE: public)



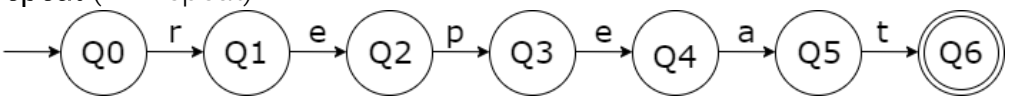
- quit (RE: quit)



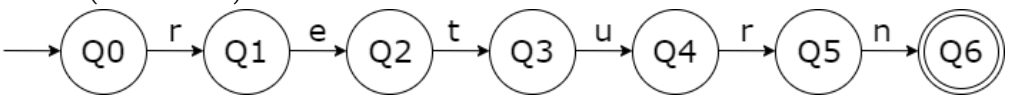
- raise (RE: raise)



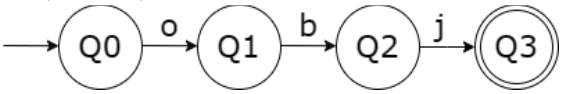
- repeat (RE: repeat)



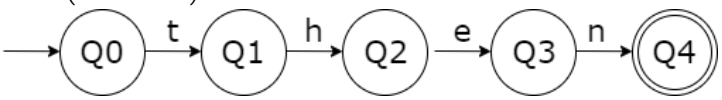
- **return** (RE: return)



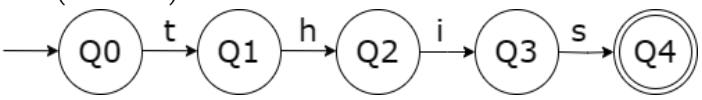
- **str** (RE: str)



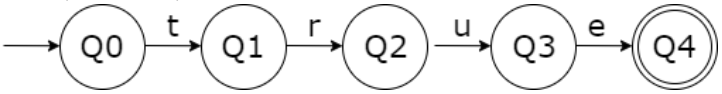
- **then** (RE: then)



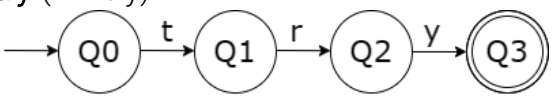
- **this** (RE: this)



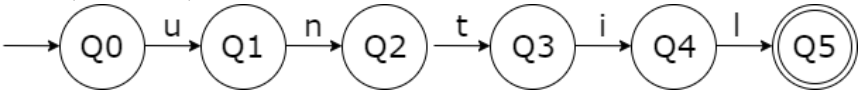
- **true** (RE: true)



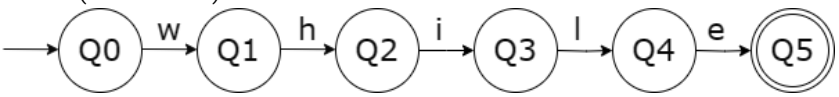
- **try** (RE: try)



- **until** (RE: until)



- **while** (RE: while)



V. NOISE WORDS

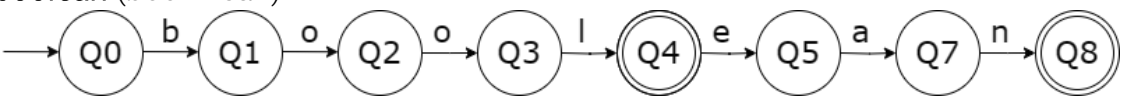
The following table contains the noise words of the luseed programming language:

Noise Words	Shorthand	Original Notation	Definition
ean	bool	boolean	A full notation for the data type "boolean", often shorthanded as "bool", making "ean" act as a noise word to enhance clarity in code and documentation.
acter	char	character	A full notation for the data type "character", often shorthanded as "char", making "acter" act as a noise word to enhance clarity in code and documentation.
ant	const	constant	A full notation for the data type "const", often shorthanded as "char", making "acter" act as a noise word to enhance clarity in code and documentation.
eger	int	integer	A full notation for the data type "integer", often shorthanded as "int", making "eger" act as a noise word to enhance clarity in code and documentation.
rmation	info	information	A full notation for the function "information", often shorthanded as "info", making "rmation" act as a noise word to enhance clarity in code and documentation.
ialize	init	initialize	A full notation for the reserved function "initialize", often shorthanded as "init", making "ialize" act as a noise word to enhance clarity in code and documentation.
ect	obj	object	A full notation for the data type "object", often shorthanded as "obj", making "ect" act as a noise word to enhance clarity in code and documentation.
ing	str	string	A full notation for the data type "string" often shorthanded as "str" making "ing" act as a noise

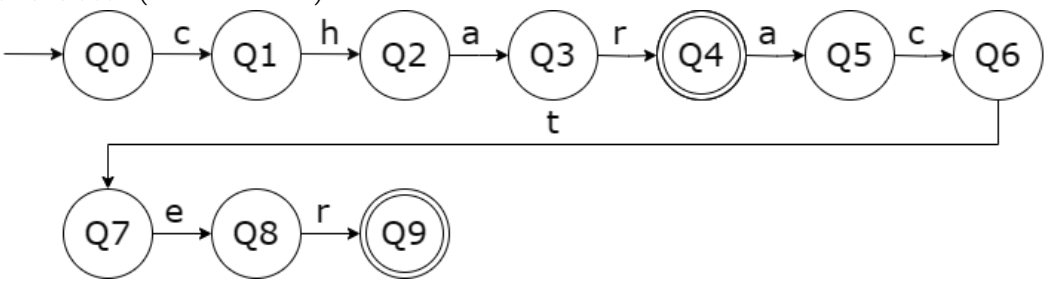
			word to enhance clarity in code and documentation.

STATE MACHINE FOR NOISE WORDS

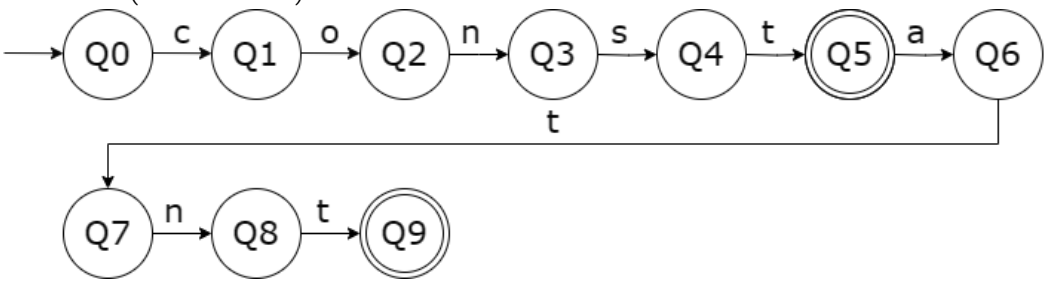
- **boolean** (bool + ean)



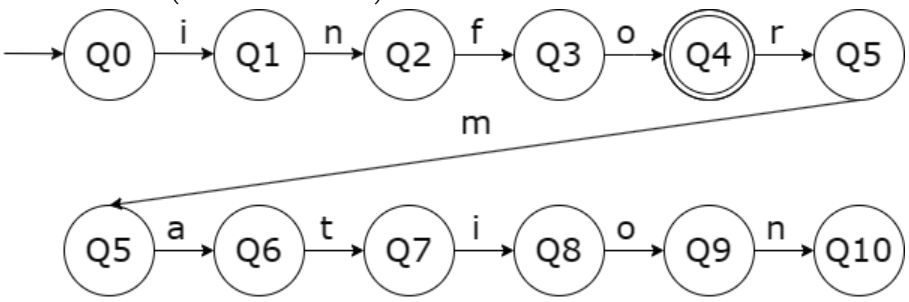
- **character** (char + acter)



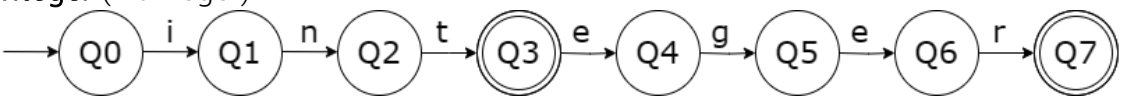
- **constant** (const + ant)



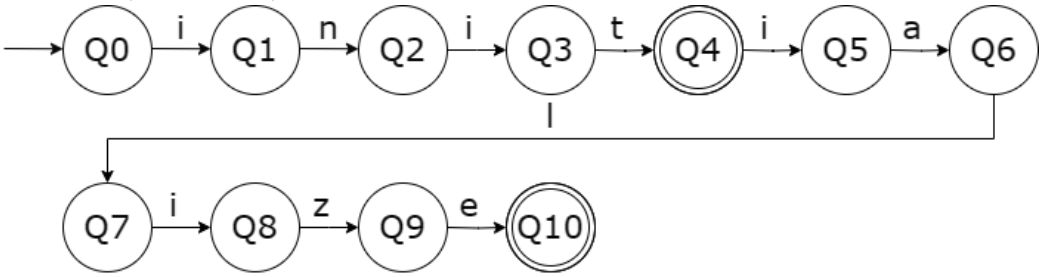
- **information** (info + rmation)



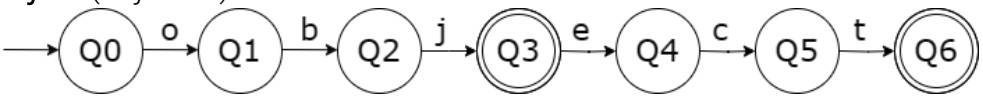
- **integer** (int + eger)



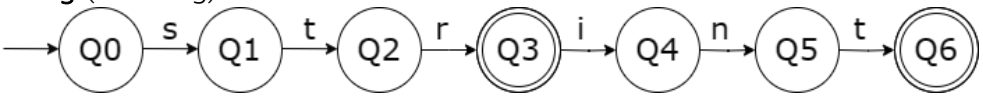
- initialize (init + alize)



- object (obj + ect)



- string (str + ing)



VI. COMMENTS

Commenting is essential for easier understandability of the source code. Almost all known programming languages known today provide the functionality of being able to annotate or comment. Thus, the luseed programming language also implemented this feature.

There are two possible ways to comment in luseed. And these are the following:

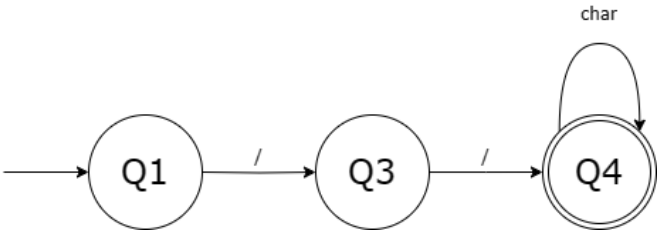
- Single-line comments - This is denoted by two forward slashes (//). Anything written between the two forward slashes and the end of the line will not be executed.

  Example:
  ```
  // This is an example comment!
  ```

- Multiple line comments – This starts with forward slash followed by an asterisk `(/*)` and ends with an asterisk and a forward slash `(*/)`. Any text that is written between `(/*)` and `(*/)` will not be executed.

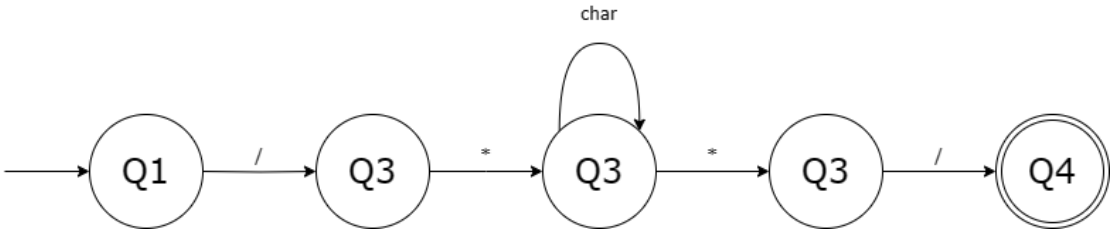
  Example:
  ```
  /* Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
  ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
  aliquip ex ea commodo consequat. */
  ```

STATE MACHINE FOR SINGLE-LINE COMMENTS



Regular Expression: (/)(/)(char)*

STATE MACHINE FOR MULTIPLE LINE COMMENTS



Regular Expression: (/)(*)(char)*(*)(/)

VII. BLANKS (Spaces)

This segment explores the blanks (spaces) in the luseed programming language which covers the importance of spaces in the language.

The following are the importance of whitespaces in the luseed programming language:

- Whitespaces (“ “) are essential in formatting code elements such as identifiers, operators, and values. Using whitespaces enhances clarity and organization of your source code making it easier to understand and debug.
- Whitespaces (“ “) can be used after keywords in order to enhance source code readability.
- With whitespaces (“ “), multiple variables can be declared in one line rather than using multiple new lines for each variable or statement.
- Code blocks enclosed within brackets in conditional or looping statements can be expressed in a single-line with the use of whitespaces (“ “).

VIII. DELIMITERS AND BRACKETS

One of the aims of luseed is to be a programming language that is easy and light for beginners, while being able to provide them with some of the common knowledge of programming languages present in the modern world. Thus, luseed uses the following delimiters and brackets not only just to tidy things up, but also to help new programmers in familiarizing the essential syntactical elements common in programming.

Delimiters

- Colon `(:)`
  - This symbol is used to end some of the built-in functions in luseed such as classes, functions, conditional statements, and iterative statements and signify that the next lines of code are included within the given statement/function.

- Semicolon `(;)`
  - Since semicolon is widely used as the statement terminator for various known programming languages such as C, C++, C#, Java, etc., it was also used as the statement terminator for the luseed programming language.

- Period `(.)`
  - Aside from being used in a floating-point value or decimals, period is also used as the object delimeter in the luseed programming language wherein it is used to access the data or property of a particular object.

- Comma `(,)`
  - In luseed programming language, comma is used as separators for different things. It can be used to separate contents of a list, the parameters of a function and many more.

Brackets

- Curly Brackets `({})`
  - These brackets are used to group blocks of code in luseed.

Example:

```
{
      int numVar = 5;
      int x;
      x += numVar;
      display(x);
}
```

- Square Brackets **([])**
  - These are used in enclosing lists and list element count in luseed.

  Example:

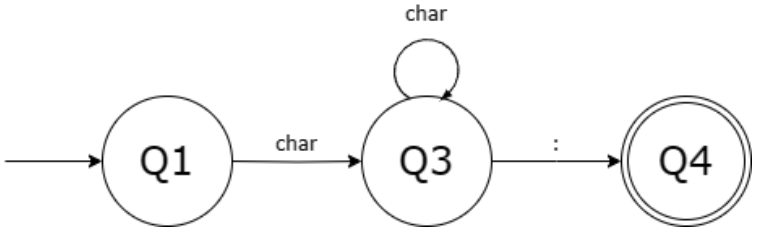
```
int numList[3] = [1, 2, 3, 4, 5];
```

- Parenthesis **(())**
  - These are used to group different expressions, as well as used in function declarations and calls.

  Example:

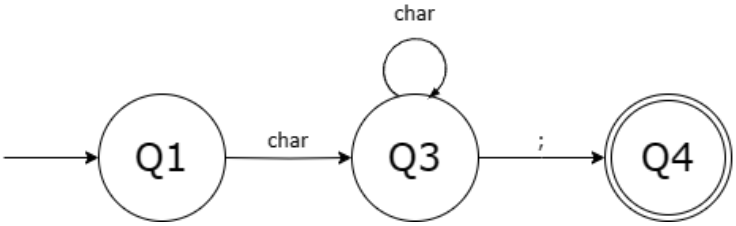
```
int x = (5 * 7 – (3 + 5));
display(“Hello World”);
```

STATE MACHINE FOR COLON



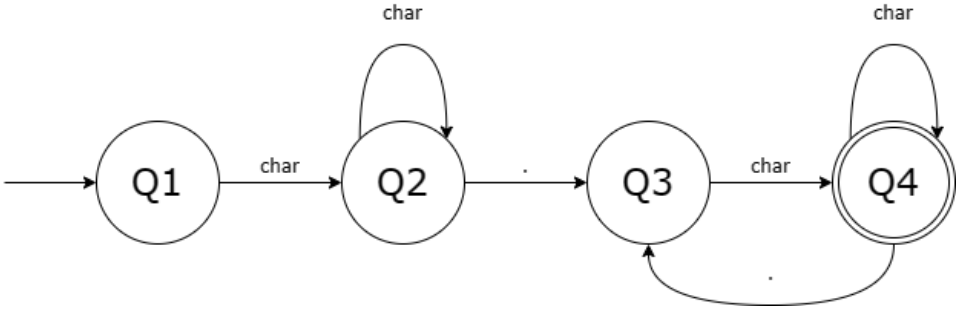
Regular Expression: (char)(char)*(:)

STATE MACHINE FOR SEMICOLON



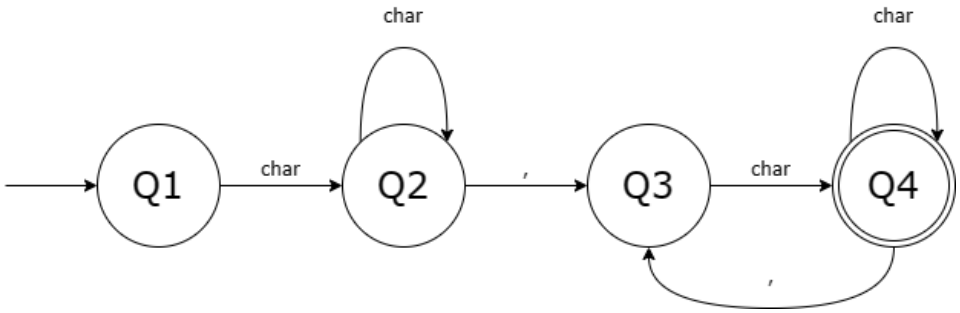
Regular Expression: (char)(char)*(;)

STATE MACHINE FOR PERIOD



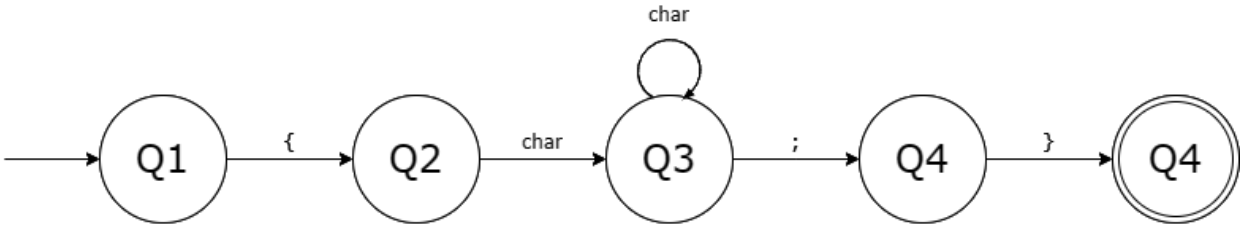
Regular Expression: (char)(char)*(.)(char)(char)*((char)(char)*(.)(char)(char)*)

STATE MACHINE FOR COMMA



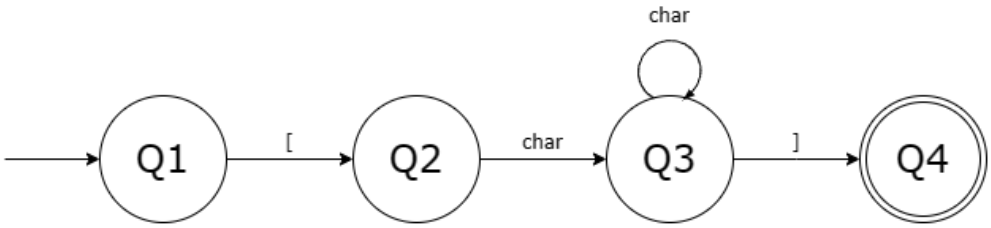
Regular Expression: (char)(char)*(,)(char)(char)*((char)(char)*(,)(char)(char)*)

STATE MACHINE FOR CURLY BRACKETS



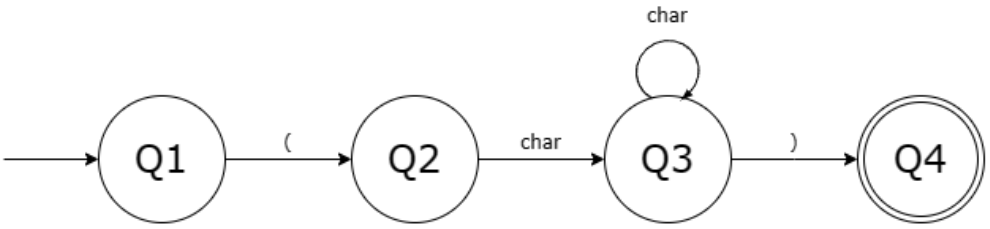
Regular Expression: ({)(char)(char)*(;)({)

STATE MACHINE FOR SQUARE BRACKETS



Regular Expression: ([)(char)(char)*(])

STATE MACHINE FOR PARENTHESIS BRACKETS



Regular Expression: (()(char)(char)*())

IX. FREE-AND-FIXED-FIELD FORMATS

This segment covers the field format of luseed programming language.

Is luseed Free-Field or Fixed-Field?

- luseed is based on multiple programming languages such as Python, Lua, and most prominently C#, which is a mix of free-field and fixed-field languages. However, the language follows a free-field format which is similar to languages such as C++, C#, and Java. The reason behind this is that the flexibility simplifies the learning experience for programming beginners by allowing code placement without strict line-by-line rules, emphasizing logical structuring over rigid formatting.

X. EXPRESSIONS

This segment of the documentation covers how the mathematical, unary, and boolean expressions is handled in luseed which includes their respective precedence, operators, descriptions, and associativity.

Essential Operators:

- OP_ARITHMETIC = {+, -, *, /, %, ~, **}
- OP_RELATION = {==, !=, >, <, <=, >=}
- OP_LOGIC = {not, or, and}

Arithmetic Expressions

- These are the expressions that return a numerical value when evaluated.

Syntax	Example
<value> <OP_ARITHMETIC> <value>	`5 + 3`
<variable> <OP_ARITHMETIC> <value>	`x - 3`
<variable> <OP_ARITHMETIC> <variable>	`intNum1 ~ intNum2`

Conditional Expressions

- Conditional expressions are the type of expressions that return either `true` or `false` when evaluated. These expressions include both relational expressions and logical expressions.
- These conditional expressions are used as the basis for the program flow in various conditional statements and looping statements.

Relational Expressions

Syntax	Example
<value> <OP_RELATION> <value>	`5 < 3`
<variable> <OP_RELATION> <value>	`varInteger_1 == 10`
<variable> <OP_RELATION> <variable>	`Num1 <= Num2`

Logical Expressions

Syntax	Example
not(<bool_value>)	`not(true)`
not(<variable>)	`not(isActive)`
<bool_value> <and>|<or> <bool_value>	`true or false`
<variable> <and>|<or> <bool_value>	`isActive and true`
<variable> <and>|<or> <variable>	`boolVal1 and boolVal2`

Definitions:

- <value> - refers to any literal value. This can either be an int, short, long, float, double, char, str, or bool value.
- <bool_value> - refers to any boolean value. This can either be `true` or `false`.
- <variable> - refers to any type of variable. This can either be an int, short, long, float, double, char, or str variable.

Take note:

- In terms of using a complex arithmetic expression, a parenthesis can be used in order to manipulate the precedence of operations.

  Example:
  ```
  10 + 5 * 6/2          will return          25
  (10 + 5) * 6/2        will return          45
  ```

- In using multiple relational or logical expressions as a condition, the user can utilize parenthesis `(())` together with the logical operators `(OP_LOGIC)`.

  Example:
  ```
  (num1 <= num2) or (1 < num3)
  (isActive or true) and (boolVal1 and not(boolVal2))
  ```

- Since both conditional expressions return a boolean value when evaluated, these conditional expressions can be used with each other. Like in the last bullet, the user

can utilize parenthesis `(())` together with the logical operators `(OP_LOGIC)` to achieve this.

Example:

```
(num1 > num2) and (isActive or isAwake)
not(boolValue) and (integerValue != num3)
```

Mathematical or Arithmetic Expressions

PRECEDENCE	OPERRATOR	DESCRIPTION	ASSOCIATIVITY
1	**	Exponent	left-to-right
2	* / ~ %	Multiplication Division Floor Division Modulus	left-to-right
3	+ -	Addition Subtraction	left-to-right

Unary Expressions

PRECEDENCE	OPERRATOR	DESCRIPTION	ASSOCIATIVITY
1	++ --	Postfix increment Postfix decrement	left-to-right
2	++ --	Prefix increment Prefix decrement	right-to-left

Conditional/Boolean Expressions (Relational and Logic)

PRECEDENCE	OPERRATOR	DESCRIPTION	ASSOCIATIVITY
1	not	Logical NEGATION	right-to-left

2	< <= > >=	Relational operator less than Relational operator less than or equal Relational operator greater than Relational operator greater than or equal	left-to-right
3	== !=	Relational operator equal to Relational operator not equal to	left-to-right
4	and	Logical AND	left-to-right
5	or	Logical OR	left-to-right
6	= += -= *= /= %= ~=	Assignment Addition Assignment Subtraction Assignment Multiplication Assignment Division Assignment Modulus Assignment Floor Division Assignment	right-to-left

XII. STATEMENTS

Statements are known as a group of expressions and/or statements that are designed to carry out a specific task or action. The next pages of the documentation contain the statements of the luseed programming language.

These statements include the following:

- Declaration Statements
- Input/Output Statements
- Assignment Statements
- Lists
- Conditional Statements
- Looping Statements
- Functions
- Classes

Declaration Statements

Types of Declaration Statements

- Unassigned Variables

  Syntax:
  `<access_modifier> <data_type> <identifier>;`

  Example:

```
public int num1;
private char sex;
protected str Name;
```

- Assigned Variables

  Syntax:
  `<access_modifier> <data_type> <identifier> = <var_value>;`

  Example:

```
protected int sum = 5 + 3;
private float piValue = x;
public str Name = “wriothesley”;
```

- Multiple Unassigned Variables

  Syntax:
  `<access_modifier> <data_type> <identifier>, ..., <identifier>;`

  Example:

```
public int num1, num2, num3;
public int val_1, val_2, val_3;
public float gravity, piValue, eValue;
```

- Multiple Assigned Variables

Syntax:
```
<access_modifier> <data_type> <identifier> = <var_value>, ...,
<identifier> = <var_value>;
```

Example:
```
public int num1 = 5, num2 = 3;
private double V_one = 1.05, V_two = 1.02;
public char person1 = ‘M’, person2 = ‘F’, person3 = ‘F’;
```

- Constant Variables

Syntax:
```
<access_modifier> const <data_type> <identifier> = <value>;
```

Example:
```
public const float PI = 3.14f;
```

Definitions:

- `<access_modifier>` - specifies the accessibility of variables, lists, functions, and classes. The `<access_modifier>`s in luseed are:
  - `public`
  - `private`
  - `protected`
- `<data_type>` - refers to the data types in luseed programming language. This can either be int, float, double, char, str, or bool.
- `<identifier>` - refers to the name that you want to supply your your objects, variables, class, etc.
- `<var_value>` - refers to the value that you want to assign to the variable. This can either be a literal value, a variable, an expression, or a function which must adhere to the declared data type.

Take Note:

- The rules in identifier naming are already listed in the document. Refer to the Identifiers page.
- In declaring multiple values, it can be a mix of assigned and unassigned variables.

  Example:

```
public int num1, num2 = 5, num3 = 10, num4;
private str person1 = “Charles”, person2, person3 = “Neuvillette”;
protected double st_1, st_2 = 90.00, st_3, st_4 = 92.50;
```

- The values assigned in the assigned variables must be similar to the data type that was provided in the beginning of the declaration statement. For example, **`int numVar`** must hold an int value.
- Variables can either have or not have access modifiers. Variables without access modifiers are **`private`** by default.

  Example:

```
str name = “Jett”;
int grade;
```

- Constant variables must always have a value and their value can never be changed.

Input Statements

Types of Input Statements:

- Input Statement without Prompt

  Syntax:
  ```
  ask();
  <variable> = ask();
  ```

  Example:
  ```
  numValue = ask();
  ```

- Input Statement with Prompt

  Syntax:
  ```
  ask(<prompt>);
  <variable> = ask(<prompt>);
  ```

  Example:
  ```
  numValue = ask(“What is your Number?”);

  public str stringValue = “What is your name?”;
  name = ask(stringValue);

  address = ask(GetAddress());
  ```

Definitions:

- `<prompt>` - this refers to the string parameter that can be passed to the `ask()` function. This is a message that will be outputted before the input. This is an optional parameter which means that the user can choose to or not to pass a prompt in the `ask()` function.

  The `<prompt>` can be one of the following:
  - A string enclosed in opening and closing double quotation marks `(“ ”)`
  - A string variable
  - A function call that returns a string value

Take Note:

- The `ask()` function must be assigned to a variable in order to store the value input by the user.
- The value that the user feeds in must have the same data type as the variable.
- The `ask()` statement should always end with a semicolon.

Output Statements

Types of Output Statements:

- Output Statement with a Display Value

  Syntax:
  `display(<display_value>);`

  Example:

```
display(“Cheescake”);
display(3);
```

  Output:

```
Cheesecake
3
```

- Output Statement with a Variable

  Syntax:
  `display(<variable>);`

  Example:

```
str cityName = “Manila”;
display(cityName);
```

  Output:

```
Manila
```

- Output Statement with Multiple Variables

Syntax:
```
display(<variable>, ..., <variable>);
```

Example:
```
int age = 5;
char sex = ‘M’;
str name = “Levi”;

display(name, sex, age);
```

Output:
```
LeviM5
```

- Output Statement with Display Value and Variable

Syntax:
```
display(<display_value>|<variable>, ..., <display_value>|
<variable>);
```

Example:
```
str StudentName = “Sthanly”;
display(“The student is ”, StudentName, “.”);
```

Output:
```
The student is Sthanly.
```

Definitions:

- **<display_value>** - refers to any type of value that a user wants to display. This can either be an int, float, double, char, str, or bool value. This can also be an expression or a function call with a returning value.

Take Note:

- In terms of displaying char values, the value must be enclosed in single quotation marks **(‘ ’)**. In displaying strings, the str value must be enclosed in double quotation marks **(“ ”)**. Lastly, in displaying int, float, double, and bool values together with variables and function calls, no quotation marks are needed to enclose the values.
- In terms of using strings as the **<display_value>**, escape sequences can be used. The escape sequences in luseed are:

Escape Sequence	Description
\’	Single quotation
\”	Double quotation
\n	New line
\\	Backslash
\t	Horizontal Tab
\v	Vertical Tab

Example:

```
display(“Hello\nWorld”);
```

Output:

```
Hello
World
```

- In displaying multiple <variable>s and displaying a mix of **<variable>**s and **<display_value>**s, it is important to use a comma **(,)** to separate **<variable>**s and **<display_value>**s.
- The **display()** statement should always end with a semicolon.

Assignment Statements

Essential Operators:

- OP_ASSIGNMENT = {=, +=, -=, *=, /=, %=, ~=}

Types of Assignment Statements:

- ### Assignment by value

  Syntax:

  ```
  <variable> <OP_ASSIGNMENT> <value>;
  ```

  Example:

  ```
  numVariable = 1;
  studentName = “Charles”;
  gravityValue = -9.81f;
  ```

- ### Assignment by variable

  Syntax:

  ```
  <variable> <OP_ASSIGNMENT> <variable>;
  ```

  Example:

  ```
  num2 += num1;
  velocity /= mass;
  shortVar2 *= shortVar3;
  ```

- ### Assignment by expression

  Syntax:

  ```
  <variable> <OP_ASSIGNMENT> <expression>;
  ```

  Example:

  ```
  Average = (90 + 90 + 98 + 93)/4;
  Sum = 5 + 3 + 10 + x + y;
  ```

```
boolVal = (x < y) and not(x);
```

Definitions:

- `<expression>` - refers to any given expression. In terms of assigning an expression to a variable, there are things that must be considered. For numerical data types (int, float, double), the type of expression that can be assigned is mathematical or arithmetic expressions and for bool data types, the type of expression must be logical or relational expressions.

Take Note:

- The equal sign `(=)` is used to assign a value to a variable.
- The plus with equal sign `(+=)` can be used with variables having numerical and string values. In numerical variables, it will perform the addition operation. On the other hand, in string variables it will perform string concatenation.
- Other `<OP_ASSIGNMENT>`, other than the equal sign `(=)` and the plus with equal sign `(+=)`, must be used with variables with numerical data types.

Lists

Types of Lists:

- One-Dimensional List

List Declaration Syntax:

```
<access_modifier> list <identifier>[<list_size>];
<access_modifier> list <identifier>[<list_size>] = [value_list];
```

Example:

```
public list floatList[3];
private list numList[3] = [1, 2, 3];
```

List Assignment Syntax (Individual):

```
<variable>[<index>] = <value>;
```

Example:

```
floatList[1] = 2f;
```

List Assignment Syntax (Entire List):

```
<variable> = [value_list];
```

Example:

```
floatList = [2f, 3f, 4f];
```

Printing of List Values Syntax:

```
display(<variable>[<index>]);
```

Example:

```
display(numList[1]);
```

Output:

```
1
```

- Multi-Dimensional List

List Declaration Syntax:

```
<access_modifier> list <identifier>[<list_size>]...[<list_size>];
<access_modifier> list <identifier>[<list_size>]...[<list_size>]
= [[<value_list>], ..., [<value_list>]];
```

Example:

```
private list charList[2][2];
protected list matrix1[2][2] = [[1, 2], [3, 4]];
```

List Assignment Syntax (Individual):

```
<variable>[<index>]...[<index>] = <value>;
```

Example:

```
charList[1][1] = ‘g’;
```

List Assignment Syntax (Entire List):

```
<variable> = [value_list];
```

Example:

```
charList = [‘c’, ‘g’, ‘f’];
```

Printing of List Values Syntax:

```
display(<variable>[<index>]...[<index>]);
```

Example:

```
display(matrix[2][1]);
```

Output:

```
3
```

Definitions:

- `<list_size>` - refers to the number of elements a list can hold. This must be an integer value.
- `<value_list>` - refers to the values of a list. This depends on the value of the `<list_size>`.
- `<index>` - refers to the position of the element that you want to manipulate or display. This must also be an integer value.

Take Note:

- A list can only hold values with different data types but can also be used like an array with only a single data type.
- The number of values that a list can hold must strictly be less than or equal to the value of the `<list_size>`.
- In luseed, the index starts at 1 rather than 0.
- Lists can either have or not have access modifiers. Lists without access modifiers are `private` by default.

Example:

```
list people[2][2];
list foods[2] = [“adobo”, “tinapay”, “kanin”];
```

Conditional Statements

Essential Operators:

- OP_RELATION = {==, !=, >, <, <=, >=}
- OP_LOGIC = {not, or, and}

Types of Conditional Statements

- If statement

Syntax:

```
if(<condition>) then: {// statement to be executed}
```

Example:

```
if(5 > 3) then: {display(“Hello”);}
```

Output:

```
Hello
```

If the statements inside the square bracket are lengthy or multiple lines of code exists inside the square brackets of the conditional statement, the following syntax can also be used:

```
if(<condition>) then:
{
      // statements to be executed
}
```

Example:

```
public float numVar1 = 5;
public float numVar2 = 10;

if(numVar1 < numVar2) then:
{
```

```
      display(numVar1, “ is less than ”, numVar2);
      display(numVar2, “ is greater than ”, numVar1);
}
```

Output:

```
5 is less than 10
10 is greater than 5
```

- If-else statement

Syntax:

```
if(<condition>) then:
{
      /* Executes the statements/s when the <condition> is
      satisfied. */
}
else:
{
      /* Executes the statements/s when the <condition> is not
      satisfied. */
}
```

Example:

```
char variable1 = “c”;
if(variable1 == “c”) then:
{
      display(“Hello World”);
}
else:
{
      display(“Goodbye World”);
}
```

Output:

```
Hello World
```

- If-else, else-if statement

Syntax:

```
if(<condition>) then:
{
      /* Executes the statements/s when the <condition> is true.
      */
}
elif(<condition>) then:
{
      /* Executes the statements/s when the <condition> in the if
      statement is false but the <condition> in elif, is true. */
}
else:
{
      /* Executes the statements/s when all <condition>s are
      false*/
}
```

Example:

```
float num1 = 10.5;
float num2 = 5.1;
if(num1 > num2) then:
{
      display(num1, " is greater than ", num2);
}
elif (num1 < num2) then:
{
      display(num1, " is less than ", num2);
}
else:
{
      display(num1, " is equal to ", num2);
}
```

Nested Conditional Statements

Conditional Statements can also exist inside the body of another conditional statement. These are called nested conditional statements. These are some of the examples:

- Nested If

Syntax:

```
if(<condition>) then:
{
      /* Executes the statements/s when the <condition> is true.
      */
      if(<ConditionalExpression>) then:
      {
            /* Executes the statements/s when the <condition>s are
            true. */
      }
}
```

Example:

```
private int score;
score = ask(“Please input your score: ”);

if(score <= 20) then:
{
      display(“Good Job!”);
      if(score < 15) then:
      {
            display(“Well done!”);
      }
}
```

Output:

```
Please input your score: 10
Good Job!
Well done!
```

- Nested If-else Statement

Syntax:

```
if(<condition>) then:
{
      /* Executes the statements/s when the
      <ConditionalExpression> is true. */
      if(<condition>) then:
      {
            /* Executes the statements/s when the last if
            statement is true and the <condition> is true. */
      }
      else:
      {
            /* Executes the statements/s when the last if
            statement is true and the <condition> is false. */
      }
}
else:
{
      /* Executes the statements/s when the <condition> in the
      first if statement is false. */
}
```

Example:

```
int userID = 1234;
int password = 5678;
int choice;
if(userID == 1234) then:
{
      if password = 5678 then:
      {
            display(“Welcome User!”);
      }
      else
      {
            display(“Wrong Password”);
      }
}
else
```

```
{
      display(“Invalid userID!”);
      choice = ask(“Do you want to try again?: );
}
```

Output:

```
Welcome User!
```

Looping Statements

Essential Operators:

- OP_RELATION = {==, !=, >, <, <=, >=}
- OP_LOGIC = {not, or, and}

Types of Looping Statements

- For Loop

  Syntax:

```
for(<init>; <condition>; <update>):
{
      /* First execute the <init> statement then check if the
      <condition> is true, execute these statements and then
      proceed to <update> and evaluate the <condition> again until
      it returns false */
}
```

  Example:

```
private list numbers[4] = [1, 2, 3, 4];

for(int i = 1; i <= 4; i++):
{
      display(numbers[i]);
}

Output:
1234
```

- For-each Loop

  Syntax:

```
foreach(<variable> in <list>):
{
      /* Execute the statements inside the loop for every
      <variable> in <list>.
```

```
}
```

Example:

```
list chars[5] = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’];

foreach(letter i in chars):
{
      display(i);
}
```

- While Loop

Syntax:

```
while(<condition>):
{
      /* If the <condition> is true, execute these statements */
}
```

Example:

```
int x = 0;
while(x < 5):
{
      display(x, “\n”);
      x++;
}
```

Output:

```
0
1
2
3
4
```

- ## Do-Until Loop

Syntax:

```
do:
{
      /* Execute these statements in the first iteration then
      evaluate the <condition>. If it is false, execute the
      statements again until the <condition> is false. */
}
until(<condition>);
```

Example:

```
int y = 0;
do:
{
      display(“Hello”);
      y++;
}
until(y > 5);
```

Output:

```
Hello
Hello
Hello
Hello
Hello
Hello
```

- ## Nested Loops

Loops can also be inside the body of other loops. This means that a for loop can be the statement that executes in a while loop. These are called nested loops. The following are some examples of nested loops in luseed programming language:

For Loop inside a For Loop

```
int i, j;
list number[2][5];
```

```
number = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]

for(i = 1; i <= 2; i++):
{
      display(“Number: ”);
      for(j = 1; j <= 5; j++):
      {
            display(number[i][j]);
            if(j == 4){
                  display(“\n”);
            }
      }
}
```

Output:

```
Number: 12345
Number: 678910
```

While Loop inside a For Loop

```
int x = 0;
int y = 0;

for(x = 0; x < 3; x++):
{
      display(x);
      while(y < 2):
      {
            display(y);
            y++;
      }
      (“\n”);
      y = 0;
}
```

Output:

```
001
```

```
101
201
```

Functions

- Main Function
  - Refers to the function where the program control starts and ends.

Syntax:

```
main():
{
      /* Statements to be executed */
}
```

Example:

```
main():
{
      display(“Hello World”);
}
```

- Function Declaration

Syntax:

```
<access_modifier> func <function_name>(<parameter_list>):
{
      /* Statements to be executed */
}
```

Example:

```
private func Multiply(int a, int b):
{
      display(a * b);
}
```

- Function Call

Syntax:

```
<function_name>(<value_list>);
```

Example:

```
Multiply(3, 5);
```

Output:

```
15
```

- Function Assignment

Function Syntax:

```
<access_modifier> func <function_name>(<parameter_list>):
{
      /* Statements to be executed */
      return <value>|<variable>;
}
```

Assignment Syntax:

```
<variable> = <function_name>(<value_list>);
```

Example:

```
public func Add(int a, int b):
{
      return a*b;
}

int num;
num = Add(20, 10);
display(num);
```

Output:

```
30
```

Definitions:

- `<function_name>` - refers to the name of the function. This must be a valid identifier.
- `<parameter_list>` - refers to the parameters needed by the function. This is optional, which means that a function can either have or not have parameter/s. The syntax for the individual parameters is the same as variable declaration but without a semicolon:

  Syntax:
  `<data_type> <identifier>`

  Example in Function Declaration:

```
func Display(int a):
{
      display(a);
}
```

  When multiple parameters are needed in a function, each parameter should be separated with a comma.

  Syntax:
  `<data_type> <identifier>, ..., <data_type> <identifier>`

  Example in Function Declaration:

```
func Display(int a, int b, int c):
{
      display(a + b + c);
}
```

- `<value_list>` - refers to the `<value>`s that the user will pass on the function in a function call. This is dependent on the number of parameters that the function accepts. If multiple parameters exist, these individual `<value>`s are separated by a comma.

Take Note:

- Functions `(func)` return any kind of data type. However, in terms of function assignment, the data type of the value that will be returned must be similar to the data type of the variable in which the value will be assigned to.

  Example:

```
private func Divide(int a, int b):
{
      int c = a ~ b;
      return c;
}

int displayNum;
displayNum = Divide(10, 3);
```

- In function assignment, it is essential that the function definition should have a return statement.
- Like variables, functions can also have or not have `<access_modifier>`s. Functions without `<access_modifier>`s are set to `private` by default.

  Example:

```
func Divide()
{
      float a, b;

}
```

Classes

- Class Declaration
  - The following contains the syntax and an example in declaring a class in luseed.

Syntax:

```
<access_modifier> class <identifier>:
{
      // Statements inside a class.
}
```

Example:

```
public class Fruit:
{
      public str name = “Apple”;
}
```

- Object Creation
  - After creating a class, an object can be created. In the example above, the Fruit class can be used in creating an object.

Syntax:

```
<access_modifier> obj <identifier> =
<class_name>(<parameter_list>);
```

Example:

```
public obj fruit = Fruit();
```

- Access Object Property
  - And object property is a variable inside a class. In accessing an object property, the following syntax can be used:

Syntax:

```
<object>.<class_property>
```

Example:

```
public obj fruit = Fruit();
display(fruit.name);
```

Output:

```
Apple
```

- The **`init`** Function
  - This is a reserved public function in a class that will always get executed when a class is initialized. It follows the following syntax:

Syntax:

```
<access_modifier> class <identifier>:
{
      init(this, <parameter_list>):
      {
            // Statements inside an initialize function.
      }
}
```

Example:

```
public class Fruit:
{
      init(this, str name, str color):
      {
            this.name = name;
            this.color = color;
      }
}

main(){
      public obj newFruit = Fruit("lime", "green");
      display(newFruit.name, "\n");
      display(newFruit.color);
}
```

Output:

```
lime
green
```

- Object Methods
  - Objects can contain functions; these functions are called methods. Here is an example using the given Fruit class above.

Example:

```
public class Fruit:
{
      init(this, str name, str color):
      {
            this.name = name;
            this.color = color;
      }

      public func DisplayName(this):
      {
            display(this.name);
      }
}
```

In calling a method, the following syntax can be used:

```
<object>.<method>(<value_list>);
```

Example:

```
public obj newFruit = Fruit();
newFruit.DisplayName(“Apple”, “Red”);
```

- this Parameter
  - The parameter this is used to provide access to variables that belong to the class.

- Modification of Object Properties
  - Values of object properties can be altered. To do that, the following syntax can be followed:

    Syntax:
    ```
    <object>.<class_property> = <value>;
    ```

    Example:
    ```
    newFruit.name = “banana”;
    ```

- Inheritance
  - Since luseed is an object-oriented language, it contains a parent class and child class.

    Parent Class
  - Any class can be a parent class; thus, the syntax is the same as declaring any class.

    Example:
    ```
    public class Person:
    {
          init(this, str name, int age):
          {
                this.name = name;
                this.age = age;
          }
    }
    ```

    Child Class
  - A child class is a type of class that inherits the functionality of another class. To do this, a parent class must be used as a parameter when creating a child class.

    Syntax:
    ```
    <access_modifier> class <identifier>(<parent_class>):
    {
          // Statements inside a class.
    ```

```
}
```

Example:

```
public class Employee(Person):
{
      pass;
}
```

A child class can also contain an **`init`** function:

```
public class Employee(Person):
{
      init(this, str name, int age):
      {
            pass;
      }
}
```

To be able to keep the **`init`** function of the parent class, include the **`init`** function of the parent class to the **`init`** function of the child class.

```
public class Employee(Person):
{
      init(this, str name, int age):
      {
            Person.init(this, str name, int age);
      }
}
```

To be able to keep all the methods of the parent class, the **`inheritall()`** function.

```
public class Employee(Person):
{
      init(this, str name, str age):
      {
            inheritall().(this, str name, int age);
      }
}
```

Lastly, child classes can have their own properties as well as methods.

```
public class Employee(Person):
{
      init(this, str name, str age):
      {
            inheritall().(this, str name, int age);
            this.company = “Google”;
      }

      func DisplayCompany(this):
      {
            display(this.company);
      }
}

obj NewPerson = Employee(“Edrick”, 18);
NewPerson.DisplayCompany();
```

Output:

```
Google
```

Definitions:

- `<class_name>` - refers to the name of the class. This must be a valid identifier.
- `<object>` - refers to an object.
- `<class_property>` - refers to a variable in a class.
- `<method>` - refers to a function inside a class.

Take Note:

- Classes, like variables, lists, and functions, can either possess or not possess an access modifier when being declared. Classes without a specific access modifier will be `private` by default.

  Example:

```
class Mamal:
{
      init(this, str name, str habitat):
      {
            this.name = name;
            this.habitat = habitat;
      }
}
```

Other Statements

- Error-Handling Statements
  - These statements are utilized for the handling of errors in luseed programming language. This includes the **`try..catch`** statement and the **`raise`** statement.

The **`try..catch`** statement

- This statement is used to handle error occurrences in a program without halting the execution process of the program.

Syntax:

```
try:
{
      // Statements to be executed.
}
catch (<Error>):
{
      // Statements to be executed.
}
```

Example:

```
try:
{
      display(arrayval[5]);
}
catch (IndexError):
{
      display(“An error has occurred”);
}
```

The way this works is that first, the code present within the **`try`** block will be executed, if an error occurs in the execution of the given code, the program will **`catch`** the **`<Error>`** and proceed to execute the code present in the **`catch`** block.

Furthermore, this can also be used with the keyword **`finally`** which allows to execute code after the **`try..catch`**, regardless of the outcome.

Syntax:

```
try:
{
      // Statements to be executed.
}
catch (<Error>):
{
      // Statements to be executed.
}
finally:
{
      // Statements to be executed.
}
```

Example:

```
try:
{
      display(arrayval[5]);
}
catch (IndexError):
{
      display(“An error has occurred!\n”);
}
finally:
{
      display(“Try-Catch Done!);
}
```

Possible Output:

```
An error has occurred!
Try-Catch Done!
```

The **raise** statement

- This statement is used to raise exceptions or errors in a program.

Syntax:

```
raise <Error>(<prompt>);
```

Example:

```
public int x = 1;

if(x > 0) then:
{
      raise Exception(“Sorry no numbers above 0”);
}
```

List of Errors

ArithmeticError	Raised when an error occurs in numerical calculations.
AttributeError	Raised when an assignment or reference to an attribute fails.
Exception	This is the base class for all exceptions.
FloatingPointError	Raised when an error occurs in calculating floating-point values.
IndexError	Raised when an index of a sequence does not exist.
KeyboardInterrupt	Raised when user presses on the Ctrl+C, Ctrl+Z, or Delete
LookupError	Base class for exceptions that pertains to something that cannot be found.
NameError	Raised when a variable does not exist.
OverflowError	Raised when the result of a numerical calculation is too large.
SyntaxError	Raised when an error in the syntax occurs.
TypeError	Raised when two different datatypes are combined.
ValueError	Raised when wrong value is in a specified data type
ZeroDivisionError	Raised when second operator of a division is zero

What's Unique?

The luseed programming language is based on various known programming languages such as C#, Lua, and Python. These include the features and principles such as the declaration, input/output, assignment, lists, conditional, looping, and function statements, together with the common syntactical elements found in the languages. However, there is no real point in creating a new language if you do not have anything new to add to the plate. Thus, the luseed programming language presents the following new features:

Beginner-Centered

Luseed is made to aid new programmers in their wonderful and exciting journey to the world of programming. That is why the syntaxes in this language are made to be humanly as possible for easier understanding and was integrated with the common syntactical elements of other programming languages to help them in familiarizing these elements for the future if they want to try out other programming languages. To specify, here are the following instances:

Input/Output Statements

Some of the famous programming languages nowadays use a very lengthy syntax just to be able to input and output a single line. This can sometimes be overwhelming for aspiring and new developers. Thus, we instead made the input and output statement as simple and clear as we can get.

For the input statement, we are inspired by the thought that every time we need an answer to specific questions or situations, we tend to ask. Thus, we set the syntax of the input function to `ask()`;. This `ask` function accepts a string parameter so that the programmers can ask for an input, as well as output a message within the same line of code. For the output statement, it came to our mind that whenever we put something on the screen, we output it or display it. With this word in mind, we made our output function syntax to `display();`. These are implemented for the purpose that it will be understandable for programmers who are not that familiar with technical terms.

To have a more detailed look, refer to the input and output statements section of the statements segment of the documentation.

Conditional Statements

In terms of the conditional statements in luseed, nothing much was changed since conditional statements in known programming languages are already very simple. However, we added the additional syntax **`then:`** after the conditional expression to provide a sentence feel to the statements.

Example:

```
if(3 < 5) then: {display(3, " is less than ", 5;} else:{display("no");}
```

For a more in-depth look, refer to the conditional statements section of the statements segment of the documentation.

Looping Statements

In luseed, two of the common looping statements such as the for loop and the while loop is present. In terms of the **`do-while`** loop, it was altered to **`do-until`** loop. Contrary to the **`do-while`** loop, the **`do-until`** loop executes the statements inside the loop body if the conditional expression returns **`false`**. Then it will proceed start a new iteration of the loop and only end when the conditional expression is met, thus the **`until`** keyword.

For a more in-depth look, refer to the looping statements section of the statements segment of the documentation.

Documentation Functionality

In line with the beginner-centered aspect of luseed, one of the unique features that the programming language possesses is a built-in documentation in a style of a tutorial system, to allow beginners grasp the technicalities of the programming language. This feature introduces `info()` function.

The info Function

- This function works similarly to the `help()` function in the Python programming language which provides the user with the information about a certain element present in the programming language. This includes the syntax of that element together with the essential definitions. Furthermore, the difference between the `info()` function and the `help()` function of python is that the output of the `info()` function will be made to be more readable as possible since the target audience of the programming language is beginner programmers.

Syntax:

```
info();
```

or

```
info(<keyword>);
```

or

```
info(<accepted_string>);
```

Examples:

The info function without parameters:

```
info();
```

Output:

```
Welcome to the information guide of luseed!

Just type the keyword or one of the accepted strings that you wish to
know about. The list of keywords and accepted strings can be found in
this link: https://luseed.com/keywords-and-accepted-strings/

If you wish to quit and go back to writing code, just type quit, and
press enter.

info >
```

The info function with <keyword> parameter:

```
info(display);
```

Output:

```
Definition:
display – is a keyword used to display/output information on the console.

Syntax:
display(<display_value>, <variable>, <func_call>);

Descriptions:
<display_value> - refers to any value that you want to put on the screen.
This can either be an int, float, double, char, str, or bool value.
<variable> - refers to any variable holding a value.
<func_call> - refers to any function call that gives off(returns) a
value.
```

```
If you want to know other statements and keywords just type info() after
the > sign below. On the other hand, type quit to quit this utility and
continue to write code.

info >
```

The info function with <accepted_string> parameter:

```
info(“input”);
```

Output:

```
In luseed, the input statement is denoted by the ask() function.

Definition:
ask – refers to the keyword used in asking for a value from the user.

Syntax:
ask(<prompt>);

Descriptions:
<prompt> - refers to the message that the program displays before asking
a value from the user. This must be a string value.

If you want to know other statements and keywords just type info() after
the > sign below. On the other hand, type quit to quit this utility and
continue to write code.

info >
```

Definitions:

- `<keyword>` - refers to the keywords present in the luseed programming language. This is used in the `info()` function to be able to know information about that keyword.
- `<accepted_string>` - refers to the string values that is accepted by both `info()` function. These refers to the various statements present in luseed programming language.

  Here is the list of `<accepted_string>`s in luseed:

Accepted String	Statement Equivalent
`“declare”`	Declaration Statements
`“input”`	Input Statements
`“output”`	Output Statements
`“assign”`	Assignment Statements
`“list”`	List Statements
`“conditional”`	Conditional Statements
`“loop”`	Looping Statements
`“function”`	Function Statements
`“class”`	Classes
`“swap”`	Swap Statements
`“loop-naming”`	Loop Naming Statements

Repeat Loop

In addition to the looping statements mentioned above, luseed introduces a new looping statement. This is good for simple usage which can be good for new programmers. This is the repeat loop.

Syntax:

```
repeat(<integer_value>):
{
      /* Execute these statements <integer_value>th number of times. */
}
```

Example:

```
repeat(3):
{
      display(“This message will display 3 times”);
}
```

Output:

```
This message will display 3 times
This message will display 3 times
This message will display 3 times
```

Definitions:

- `repeat loop` - is a simple kind of looping statement that executes the statement/s in a loop body within a given number of times. This is a variant of a for loop but with a constant initialization statement value `(1)` and update statement `(increment)`.
- `<integer_value>` - refers to the number of times the given statement/s within a loop body are executed. This must strictly be an integer value.

Swap Function

Luseed introduces a built-in function called the `swap()` function. This allows programmers to swap values of two or three different variables with the same data type.

Two Variable Swap

- In swapping two variables, the following syntax is followed:

  Syntax:
  `swap(<variable_1>, <variable_2>);`

  Example:

```
public int a = 5;
public int b = 10;

display(a, “\t”, b, “\n”);
swap(a, b);
display(a, “\t”, b, “\n”);
```

  Output:

```
5     10
10    5
```

  In this example, the values of int a which is 5 in the declaration statement and int b which is 10 in the declaration statement are swapped using the `swap()` function.

Three Variable Swap

- In swapping three variables the following syntax is followed:

  Syntax:
  `swap(<variable_1>, <variable_2>, <variable_3>);`

Example:

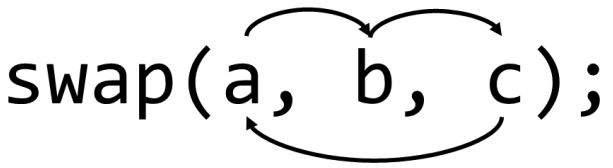
```
int a = 5;
int b = 10;
int c = 15;

display(a, “\t”, b, “\t”, c);
swap(a, b, c);
display(a, “\t”, b, “\t”, c);
```

Output:

```
5     10    15
15    5     10
```

In swapping three values, nothing much is changed in terms of the syntax, it is pretty much the same with swapping two variables. However, the logic behind the swap is different.

The way the swap works is that it swaps in a clockwise manner where the value of the leftmost variable will be set to the second variable on the right and the value of the second variable will be set to the third variable and the value of the third variable will be set to the first. To have a visual understanding refer to the figure below:



In the given example above, let us say that you want to give the value of b to a, you must rewrite the statement to this:

```
swap(a, c, b);
```

Loop Naming

Luseed also introduces loop naming. This is a coding principle that involves assigning a unique identifier, or "loop name," to a loop structure. This practice enhances control over loops, leading to more readable and understandable code.

To declare a desired name for a loop, simply type the `loop` keyword and then input the loop name along with a colon symbol `(:)`, before initializing the looping statement.

For Loop

Syntax:

```
loop <identifier>: for(<init>; <condition>; <update>):
{
      // loop body
}
```

For-each Loop

```
loop <identifier>: foreach(<variable> in <list>):
{
      // loop body
}
```

While Loop

Syntax:

```
loop <identifier>: while(<condition>):
{
      // loop body
}
```

Do-Until Loop

```
loop <identifier>: do:
{
      // loop body
}
until(<condition>);
```

Repeat Loop

```
loop <identifier>: repeat(<integer_value>):
```

```
{
      // loop body
}
```

Example:

```
loop outer_loop: repeat(10):
{
      display(“Hello”);
}
```

This can be helpful in manipulating looping statements. One example application is breaking a nested looping statement in an inner loop.

Example:

```
loop outerloop: for(int i = 1; i < 5; i++):
{
      loop innerloop: for(int j = 1; j < 5; j++):
      {
            if(j > 3) then:
            {
                  break <outerloopname>;
            }
      }
}
```

In the example above, when the conditional statement inside the `innerloop` is `true`, the `break` statement will be called which will then terminate the `outerloop`. This means that the entire nested loop will be terminated, rather than just the `innerloop`.

Furthermore, these following statements can be used together with the loop name:

- `break`
- `continue`
- `pass`

Check Function

The luseed programming language also offers another new function that can be quite beneficial for error management and debugging in the context of classes. This is called the `check()` function. This function is used to check whether a certain property or method is present within a given class. Similar to the `try..catch` statement, this function will not halt the execution of the program when no such property or method is found.

Check Function for Class Property

- In checking whether a property is present within a class, the following syntax is followed:

  Syntax:
  ```
  check(<class_property>);
  ```

  This statement returns the following:
  - If the property exists within the class, the built-in function will return a list containing the following:

    ```
    [true, <class_property_value>]
    ```

  - If the property does not exist within the class, the built-in function will display a prompt, as well as return a list:

    ```
    The property does not exist within the class <class_name>.
    [false, null]
    ```

  Example:

```
public class Foo:
{
      public str name = “LastName”;
}

main():
{
      list classprop1 = check(Foo.name);
      display(classprop1, “\n”);
      list classprop2 = check(Foo.age);
```

```
     display(classprop2);
}
```

Output:

```
[true, “LastName”]
The property does not exist within the class Foo.
[false, null]
```

Check Function for Class Method

- In checking whether a method is present within a class, the following syntax is followed:

  ```
  check(<class_method>());
  ```

  This statement returns the following:

  - If the property exists within the class, the built-in function will return a list containing the following:

    ```
    [true, <method_return_value>]
    ```

    Note: When the method does not have any return value, it will instead return a `null`.

  - If the property exists within the class but the function requires parameter/s and the function call present in the `check` function does not pass any, the built-in function will display a prompt, as well as return a list:

    ```
    The method exists within the class <class_name> but requires the
    following parameters: <parameter_list>.
    [true, null]
    ```

  - If the method does not exist within the class, the built-in function will display a prompt, as well as return a list:

    ```
    The method does not exist within the class <class_name>.
    [false, null]
    ```

Example:

```
public class Foo:
{
      public func TestProperty1():
      {
            return “Hello World”
      }

      public func TestProperty2(str name):
      {
            return name;
      }
}

main():
{
      list list1 = check(Foo.TestProperty1());
      display(list1, “\n”);
      list list2 = check(Foo.TestProperty2());
      display(list2, “\n”);
      list list3 = check(Foo.TestProperty3(“Raiden”));
      display(list3, “\n”);
}
```

Output:

```
[true, “Hello World”]
The method exists within class Foo but requires the following
parameters: str name.
[true, null]
The method does not exist within the class Foo.
[false, null]
```

Take Note:

- This function can be used in debugging class methods and properties.
- This new function solves the problem of the `hasattr()` function of python, wherein it can only identify whether a property or a method exists within a class but not able to access it. Thus, the reason why `check()` function was created.
- It lessens the space consumption of the source code file since it only takes one line.

For instance, suppose that the luseed programming language has an `hasattr()` function the code for checking whether a method or property is present within a class will be the following:

In this case we will be checking for a method present in a class.

```
if hasattr(Foo, “age”):
{
      try:
            Foo.age()
      catch (AttributeError):
            display("Missing parameters")
}
else:
{
      display("Function does not exist")
}
```

It might not be much right now, but as you can see, using this way of identifying whether a method exists within a class and acquiring its value consumes more lines which can take up more space for the source code, especially when multiple methods/properties are to be tested. So instead of using multiple lines for that, we can use this statement instead:

```
check(Foo.age());
```

It is simpler, takes less space consumption in the long run, and will be easy for beginners to understand.

- The `check()` function can also be beneficial in demonstrating an aspect of object-oriented programming. More specifically, it can help provide an idea about inheritance wherein the `check()` function can be used to a child class to find out whether it inherited the method/property of the parent class.