University of the Philippines Los Banos
College of Arts and Sciences
Institute of Computer Science

CMSC 124

Design and Implementation of Programming Languages

PROJECT SPECIFICATIONS

First Semester A.Y. 2025-2026

Specifications based on
KBPPelaez' and CNMPeralta's Previous CMSC124 Project Specifications

General Instructions

You are to create an **interpreter** for the LOLCode Programming Language. The constructs required for the project are discussed in the succeeding pages of this document. However, more information regarding LOLCode can be found here: [Website] [Original Specifications] [Interpreter].

The project-making process is divided into 2 phases:

1. **Planning Phase**
   In this phase, you are to submit [type]written documents that you will use during the coding phase. These documents must be submitted early so that we can annotate them with corrections if there are any.

   The documents are as follows:
   a. **Patterns for LOLCode Lexemes**
      A list of regular expressions that will match the lexemes of the LOLCode PL must be submitted by September 29. Answers must be written/typed using this template: [Google Docs].
   b. **Grammar for LOLCode Syntax**
      The grammar that your LOLCode interpreter will follow must be written/typed using this template: [Google Docs]. This must be submitted by October 23.

2. **Coding Phase**
   You must start coding at least a month before the end of the classes. You are to present your progress to your lab instructors three (3) times: one for the lexical analyzer part, one for the syntax analyzer part, and one for the semantics of your code.

   The progress presentations are required *and are graded* to guarantee that you are coding the interpreter correctly, thus lessening the possibility of re-coding in the succeeding presentations.

You are allowed to start coding once the feedback on your regular expressions has been given.

Note!

You are required to submit a working interpreter. At the minimum, your interpreter must be able to evaluate at least one operation or statement.

Since you are to create an interpreter, the interpreter must be able to analyze each line of the program *lexically, syntactically, and semantically*. If your program cannot completely execute these three phases of interpretation, that will not be accepted, and thus you will be given a grade of 0 for the coding portion of your project.

Specifications

The minimum requirements for the project are listed below.

FILE NAMING AND FORMATTING <<

- LOLCode source files should have the `.lol` file extension.
- LOLCode programs should start with the `HAI` keyword (nothing before, except comments or functions (if you implement functions)), and end with the `KTHXBYE` keyword (nothing after, except comments or functions (if you implement functions)).
- There is no need to include the version number after the `HAI` keyword.

sample1.lol	**sample2.lol**
HAI *BTW statements here* KTHXBYE	*BTW this is accepted!* HAI *BTW statements here* KTHXBYE *BTW this is accepted!*

SPACING/WHITESPACES <<

- You may assume that one line contains one statement only. There is no need to support soft command breaks. Each statement is delimited by the new line.

```
HAI
  BTW no need to support this
  I HAS A ...
  var ITZ 2, VISIBLE var
KTHXBYE
```

- You may assume that there is only one whitespace between keywords.

`I HAS A var1 BTW YEEES!`	`I    HAS A       var2 BTW NO`

- Indentation is irrelevant.
- Spaces inside a `YARN` literal should be retained.

```
“Spaces          between” BTW should NOT become “Spaces between”
```

COMMENTS <<

- Comments are not considered statements, and must be ignored. They should be able to coexist with another statement on a line.

```
HAI
  I HAS A var ITZ 2 BTW I’m allowed!
KTHXBYE
```

- Keywords `OBTW` and `TLDR` for multi-line comments must have their own lines, i.e., they cannot co-exist with other statements. The `OBTW` and `TLDR` must have their own lines (which may include some comments *but not other statements*).

```
I HAS A var ITZ 2 OBTW Hi! TLDR        ← NOT ALLOWED!!!
I HAS A var OBTW noot TLDR ITZ 2       ← NOT ALLOWED!!!
I HAS A num OBTW konnichiwa            ← NOT ALLOWED!!!
            TLDR

I HAS A var1                           ← NOT ALLOWED!!!
```

```
OBTW what way?
TLDR I HAS A var2

I HAS A var3                        ← YASSSS ALLOWED!!!
OBTW this
     Way
TLDR
```

VARIABLES <<

- All variables must be declared inside the WAZZUP portion of the program. The WAZZUP clause is found at the beginning of the program, after HAI.

```
WAZZUP
       BTW Declare variables here
BUHBYE
```

- Variable names must start with a letter, followed by any combination of letters, numbers, and underscores. No spaces, dashes, or other special symbols are allowed to be part of the variable name.
- Variable declaration is done using the keyword I HAS A.
- Variable initialization is done using the ITZ keyword, and the value may be a literal, the value of another variable, or the result of an expression. Initializing variables is not required.

```
I HAS A thing                  BTW uninitialized var
I HAS A thing2 ITZ “some”      BTW literal
I HAS A thing3 ITZ thing2      BTW variable
I HAS A thing4 ITZ SUM OF 5 AN 4   BTW expression
```

- Initialization may contain any operation that results to a literal value.
- You should be able to implement the implicit IT variable.

DATA TYPES <<

- Variables in LOLCode are dynamically-typed, i.e., the data type of its variables changes automatically when a new value of a different data type is assigned.
- LOLCode implements the following data types:

DATA TYPE	IN LOLCode	DESCRIPTION
untyped	NOOB	The data type of uninitialized variables is NOOB.
integer	NUMBR	These are sequences of digits without a decimal point (.) and are not enclosed by double quotes. Negative numbers must be preceded by a negative sign (-), but positive numbers **MUST NOT** be preceded by a positive sign (+).
float	NUMBAR	They are sequences of digits with exactly one decimal point (.) and are not enclosed by double quotes. They may be preceded by a negative sign (-) to indicate that the value is negative. For positive values, it **MUST NOT** be preceded by a positive sign (+) to indicate that it is positive.
string	YARN	These are delimited by double quotes (“”).
boolean	TROOF	The value of a TROOF can be WIN (true) or FAIL (false).

- Special characters inside YARNs (e.g. :), :>, etc.) are not required.

OPERATIONS <<

- Operations are in prefix notation.
- All operations except SMOOSH, ALL OF, ANY OF, and NOT are binary.
- SMOOSH, ALL OF, and ANY OF are of infinite arity.
- NOT is unary.
- All operations except SMOOSH, ALL OF, and ANY OF can be nested.
- The AN keyword is required to separate operands.

Input/Output

- Printing to the terminal is done using the VISIBLE keyword.
- VISIBLE has infinite arity and concatenates all of its operands after casting them to YARNs. Each operand is separated by a '+' symbol.
- The VISIBLE statement automatically adds a new line after all the arguments are printed.
- Accepting input is done using the GIMMEH keyword.
- GIMMEH must always use a variable, where the user input will be placed. The input value is always a YARN.

Arithmetic/Mathematical Operations

- Below are the arithmetic operations:

```
SUM OF <x> AN <y>            BTW + (add)
DIFF OF <x> AN <y>           BTW - (subtract)
PRODUKT OF <x> AN <y>        BTW * (multiply)
QUOSHUNT OF <x> AN <y>       BTW / (divide)
MOD OF <x> AN <y>            BTW % (modulo)
BIGGR OF <x> AN <y>          BTW max
SMALLR OF <x> AN <y>         BTW min
```

- Mathematical operations are performed with NUMBRs and/or NUMBARs involved.
- If a value is not a NUMBAR and is not a NUMBR, it must be implicitly typecast into a NUMBAR/NUMBR depending on the value. Typecasting rules must apply (see section on typecasting).
- If a value cannot be typecast, **the operation must fail with an error**.
- If **both** operands evaluate to a NUMBR, the result of the operation is a NUMBR.
- If **at least one** operand is a NUMBAR, the result of the operation is a NUMBAR.
- Nesting of operations is allowed, but all operations are still binary.

```
SUM OF 2 AN 4               BTW result is NUMBR
DIFF OF 4 AN 3.14           BTW result is NUMBAR
PRODUKT OF "2" AN "7"       BTW result is NUMBR
QUOSHUNT OF 5 AN "12"       BTW result is a NUMBR
MOD OF 3 AN "3.14"          BTW result is a NUMBAR
SUM OF QUOSHUNT OF PRODUKT OF 3 AN 4 AN 2 AN 1    BTW ((3*4)/2)+1
SUM OF SUM OF SUM OF 3 AN 4 AN 2 AN 1             BTW ((3+4)+2)+1
```

Concatenation

- The syntax for string concatenation is shown below:

```
SMOOSH str1 AN str2 AN ... AN strN  BTW str1+str2+...+strN
```

- SMOOSH does not require the MKAY keyword.
- If the operand evaluates to another data type, they are implicitly typecast to YARNs when given to SMOOSH. For example, 124 will become "124", 2.8 will become "2.8", WIN will become "WIN", and so on.

Boolean Operations

- Below are the boolean operations:

```
BOTH OF <x> AN <y>                BTW and
EITHER OF <x> AN <y>              BTW or
WON OF <x> AN <y>                 BTW xor
NOT <x>                           BTW not
ALL OF <x> AN <y> ... MKAY        BTW infinite arity AND
ANY OF <x> AN <y> ... MKAY        BTW infinite arity OR
```

- If the operands are not TROOFs, they should be implicitly typecast.
- ALL OF and ANY OF cannot be nested into each other and themselves, but may have other boolean operations as operands.

```
ALL OF NOT x AN BOTH OF y AN z AN EITHER OF x AN y MKAY
BTW (!x) ∧ (y∧z) ∧ (x∨y) ← YAASSS ALLOWED!!
ALL OF ALL OF x AN y MKAY AN z MKAY BTW :( ← NOT ALLOWED!!
```

Comparison Operations

- Below are the comparison operations:

```
BOTH SAEM <x> AN <y>             BTW x == y
DIFFRINT <x> AN <y>              BTW x != y
```

- Relational operations are created by adding the BIGGR OF or SMALLR OF operations:

```
BOTH SAEM <x> AN BIGGR OF <x> AN <y>     BTW x >= y
BOTH SAEM <x> AN SMALLR OF <x> AN <y>    BTW x <= y
DIFFRINT <x> AN SMALLR OF <x> AN <y>     BTW x > y
DIFFRINT <x> AN BIGGR OF <x> AN <y>      BTW x < y
```

- Comparisons are done using integer math if the operands are NUMBRs, and floating-point math if the operands are NUMBARs.
- There is no automatic typecasting for operands in a comparison operation.

Typecasting

- The following are the rules you must follow when typecasting in LOLCode.

DATA TYPE	DESCRIPTION
NOOB	• NOOBs can be implicitly typecast into TROOF. Implicit typecasting to any other type except TROOF will result in an error. • Explicit typecasting of NOOBs is allowed and results to empty/zero values depending on the type.
TROOF	• The empty string (“”) and numerical zero values are cast to FAIL. • All other values, except those mentioned above, are cast to WIN. • Casting WIN to a numerical type results in 1 or 1.0. • Casting FAIL results in a numerical zero.
NUMBAR	• Casting NUMBARs to NUMBR will truncate the decimal portion of the NUMBAR. • Casting NUMBARs to YARN will truncate the decimal portion up to two decimal places.
NUMBR	• Casting NUMBRs to NUMBAR will just convert the value into a floating point. The value should be retained. • Casting NUMBRs to YARN will just convert the value into a string of characters.
YARN	• A YARN can be successfully cast into a NUMBAR or NUMBR if the YARN does not contain any non-numerical, non-hyphen, non-period characters.

- Explicit typecasting a value can be done using the `MAEK` operator. This operator, however, only modifies the resulting value, and not the variable involved, if there is any.

```
I HAS A var1 ITZ 12     BTW var1 is a NUMBR
MAEK var1 A NUMBAR      BTW returns NUMBAR equivalent of var1 to IT (12.0)
                        BTW var1 is still a NUMBR
MAEK var1 YARN          BTW returns YARN equivalent of var1 to IT (“12”)
                        BTW var1 is still a NUMBR
```

- Re-casting a variable can be done via normal assignment statement involving `MAEK` or via `IS NOW A`.

```
I HAS A number ITZ 17        BTW number is NUMBR type
number IS NOW A NUMBAR       BTW number is NUMBAR type now (17.0)
number R MAEK number YARN    BTW number reassigned to the YARN value of number (“17.0”)
```

STATEMENTS <<

Expression Statements

- The result of an expression may not be assigned to a variable. In this case, its result will be stored in the implicit variable `IT`.

Assignment Statements

- The assignment operation keyword is `R`.
- The left-hand side is always a receiving variable, while the right side may be a literal, variable, or an expression.

```
<variable> R <literal>
<variable> R <variable>
<variable> R <expression>
```

Flow-control Statements

- LOLCode has two kinds of conditional statements: if-then and switch-case.

IF-THEN STATEMENTS

- The IF-THEN statement in LOLCode uses five keywords: `O RLY?`, `YA RLY`, `MEBBE`, `NO WAI`, and `OIC`. The syntax for if-then statements is shown below:

```
<expression>                BTW result is stored in IT
O RLY?
   YA RLY                   BTW if
      <if code block>
   NO WAI                   BTW else
      <else code block>
OIC
```

- Indentation is irrelevant.
- If the `IT` variable can be cast to `WIN`, the if-clause executes. Otherwise, the else-clause executes.
- Implementing `MEBBE` (else-if) clauses is not required.
- The if-clause starts at the `YA RLY` keyword and ends when the `NO WAI` or `OIC` keyword is encountered.
- The else-clause starts at the `NO WAI` keyword and ends when the `OIC` keyword is encountered.
- You may assume that `O RLY?`, `YA RLY`, `NO WAI`, and `OIC` are alone in their respective lines. `MEBBE`, if implemented, should be followed by an expression in the same line.

SWITCH-CASE STATEMENTS

- There are four (4) keywords used in a switch-case in LOLCode: `WTF?`, `OMG`, `OMGWTF`, and `OIC`. The syntax for switch-case statements is shown below:

```
WTF?                    BTW uses value in IT
OMG <value literal>
   <code block>
[OMG <value literal>
   <code block>...]
[OMGWTF
   <code block>]
OIC
```

- Once `WTF?` is encountered, the value of the implicit `IT` variable is compared to each case, denoted by an `OMG` keyword. If `IT` and the case are equal, the succeeding code block executes until a `GTFO` (break) or an `OIC` keyword is encountered.
- The cases may be of any literal type (`NUMBR`s, `NUMBAR`s, `YARN`s, and `TROOF`s).
- The default case is specified by `OMGWTF` and is executed if none of the preceding cases match the value of `IT`. Execution then stops when an `OIC` is encountered.

LOOPS

- Loops in LOLCode follow the form below:

```
IM IN YR <label> <operation> YR <variable> [TIL|WILE <expression>]
    <code block>
IM OUTTA YR <label>
```

- The `IM IN YR <label>` and `IM OUTTA YR <label>` clauses specify the start and end of the loop. The `<label>` follows the format for a valid variable name, and is used as a delimiter, especially in the case where nested loops are implemented.
- The `<operation>` can either be `UPPIN` (increment by 1) or `NERFIN` (decrement by 1), which modifies the `<variable>` that follows.
- The variable specified in `<variable>` should be an existing variable (i.e., declared) and whose value can be cast to a numerical value so it can be processed by `UPPIN`/`NERFIN`.
- The loops can be terminated by meeting the condition expressions `TIL`/`WILE` or by issuing a `GTFO` statement inside the loop.
- The `TIL <expression>` clause will repeat the loop as long as `<expression>` is `FAIL`.
- The `WILE <expression>` clause will repeat the loop as long as `<expression>` returns `WIN`.

```
I HAS A temp ITZ 2
BTW prints 2 to 9 using TIL
IM IN YR print10 UPPIN YR temp TIL BOTH SAEM temp AN 10
   VISIBLE temp
IM OUTTA YR print10

BTW at this point, temp's value is 10, so we must reassign its initial value
temp R 2

BTW prints 2 to 9 but using WILE
IM IN YR print10 UPPIN YR temp WILE DIFFRINT temp AN 10
   VISIBLE temp
IM OUTTA YR print10
```

FUNCTIONS <<

Definition

- Functions have a fixed number of parameters in the definition.

```
HOW IZ I <function name> [YR <parameter1> [AN YR <parameter2> [AN YR <parameter3> ...]]]
```

```
    BTW function body
  IF U SAY SO
```

- `<parameter1>`, `<parameter2>`, and `<parameter3>` are the parameters of the function. If there are no parameters, then the parameters will be ommitted:

```
  HOW IZ I sample_function                  BTW function with 0 arguments
  HOW IZ I sample_function2 YR x AN YR y    BTW function with 2 arguments
```

- Functions cannot access identifiers outside of it. Only the arguments passed are accessible to the function.
- Arguments are passed via pass-by value only.
- The parameters in the function become a variable of that function, with the passed argument as the initial value.

Returning

- `FOUND YR <expression>` returns the value of the expression.
- `GTFO` returns with no value (NOOB), but if no `GTFO` is found, the return type will also be NOOB automatically.
- Return values will automatically be stored in the implicit variable `IT`.

Calling

- Functions can be called using the following syntax:

```
  I IZ <function name> [YR <expression1> [AN YR <expression2> AN YR <expression2>]] MKAY
```

- The expressions must be executed first before executing the function body.

>> <<

The specifications listed here must be followed first. For any other rules not specified here, you may follow the specifications from Github.

Extra Credit

Implementing anything that is not required in the project will give you bonus pBTWoints. The number of bonus points depends on the level of difficulty of the feature you implement. Possible bonus features are:

Soft-line/command breaks (,)	Special characters in a YARN (:>, :), etc)	Loop-nesting	MEBBE
Special characters in strings	Line continuation (...)	Else-if clauses	Array
Suppress the newline after a line of output by ending the VISIBLE statement with a !			
Nesting of different flow control statements		If-else nesting	Switch-nesting

Scoring

Since this is a group project, there will be a peer evaluation at the end of the semester. The peer evaluation score will be directly multiplied by your group's overall score in the project. You will evaluate each of your group mates AND yourself. Refer to the example below:

MEMBER	GROUP SCORE	PEER EVAL (Average)	FINAL PROJECT GRADE
Carlo	97.63%	100%	97.630
Beili		80%	78.104
Erika		60%	58.578

The peer evaluation has a big effect on your final project grade. **Be mindful of the contributions you make to the group project and always put your best foot forward** so that your group mates will want to give you the full 100% for their evaluation of your contributions, cooperativity, etc.

The breakdown for computing the project group score is as follows:

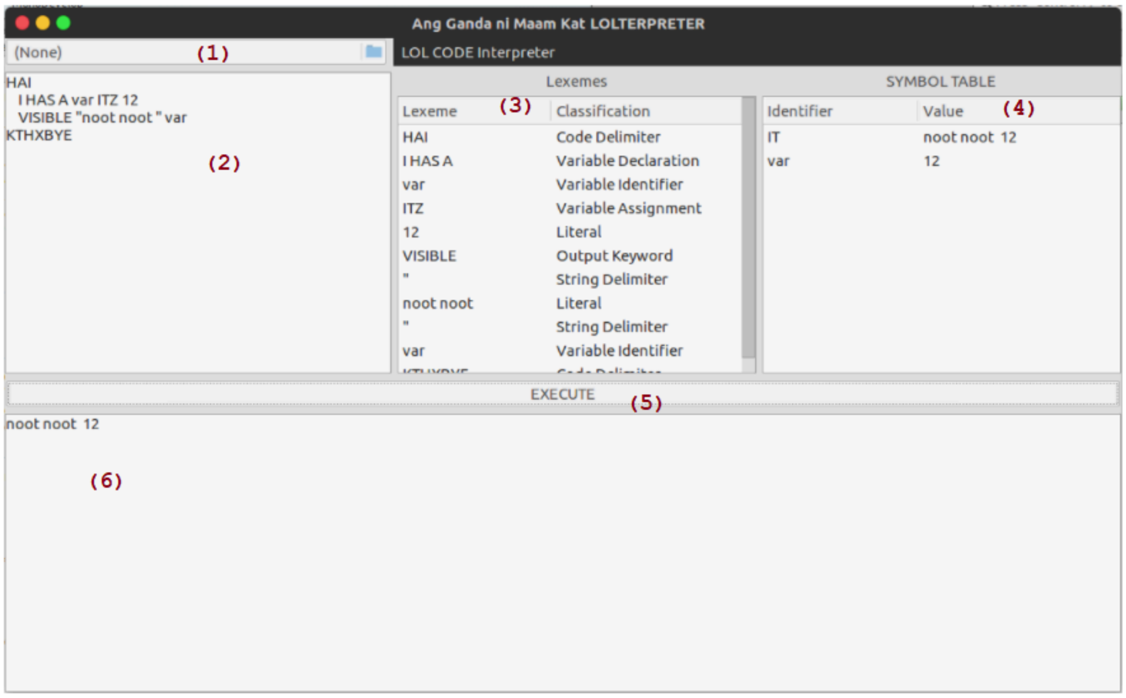
SUB-UNIT		**POINTS**
Regular Expressions		10
Grammars		10
User Input	GIMMEH	5
User Output	string/literal	1
	variable	2
	expression	3
Variables	I HAS A	5
	ITZ literal	2
	ITZ expr	3
If-Else	YA RLY	3
	NO WAI	3
Milestone Presentations (5 points each)		15
	TOTAL	**62**

+

SUB-UNIT		**POINTS**
Operations	Assignment	2
	Arithmetic	3
	Comparison	3
	Boolean	3
Typecasting		5
Switch-Case	OMG	3
	OMGWTF	3
	GTFO	3
Loops	Delimiter	2
	UPPIN/NERFIN	3
	TIL	4
	WILE	4
Functions	Definition	3
	Return	3
	Calling	2
	TOTAL	**46**
	GRAND TOTAL	**108**

Submission Format

- **You may use any programming language** that you want.
- A **Graphical User Interface (GUI) is required** and should look similar to the diagram below:



```
HAI
  I HAS A var ITZ 12
  VISIBLE "noot noot " var
KTHXBYE
```

Lexeme	Classification
HAI	Code Delimiter
I HAS A	Variable Declaration
var	Variable Identifier
ITZ	Variable Assignment
12	Literal
VISIBLE	Output Keyword
"	String Delimiter
noot noot	Literal
"	String Delimiter
var	Variable Identifier

Identifier	Value
IT	noot noot 12
var	12

```
noot noot 12
```

- The parts of the GUI are as follows:
  - **(1) File explorer –** Allows you to select a file to run. Once a file is selected, the contents of the file should be loaded into the text editor (2).
  - **(2) Text editor –** Allows you to view the code you want to run. The text editor should be editable, and edits done should be reflected once the code is run.
  - **(3) List of Tokens –** This should be updated every time the Execute/Run button (5) is pressed. This should contain all the lexemes detected from the code being ran, and their classification.
  - **(4) Symbol Table –** This should be updated every time the Execute/Run button (5) is pressed. This should contain all the variables available in the program being ran, and their updated values.
  - **(5) Execute/Run button –** This will run the code from the text editor (2).
  - **(6) Console –** Input/Output of the program should be reflected in the console. For variable input, you can add a separate field for user input, or have a dialog box pop up.
- For (3) and (4) in the GUI, you can either: a) update the list/table every time a line is read, or (b) update the list of tokens one time, and the symbol table every time a value is updated.
- You are **NOT ALLOWED to use Flex/Lex or YACC/Bison or Parsing Expression Grammars (PEG) or any lexer/parser generator tools**. You are required to implement your own lexical and syntax analyzer.

- The zipped file of your codebase will be uploaded in the classroom.
- A document containing detailed instructions for submission and presentation will be posted separately.