Programming Project

Custom Programming Language Implementation

OBJECTIVE

To be able to implement a simple compiler for a custom programming language.

INSTRUCTIONS

1. You are to work on this activity by group.
2. You are to write a code for a program that follows the details in the PROGRAM SPECIFICATION section. There should only be one program for this PE.
3. You are to develop your program following the structured programming approach at the very least.
4. All your program files (source code files, etc.) should be together with your main program file (the one that contains the main function). There should be no need to put certain files in certain folders just so your program will compile and run.
5. Follow the details specified in the SUBMISSION REQUIREMENT section.

PROGRAM SPECIFICATION

THE LANGUAGE

The name of the programming language is Integer-Oriented Language (IOL). IOL is a simplified custom language that only involves integer-type values as the numeric values and numerical operations and expressions. The following set of specifications covers the rules that should be observed and implemented in using the language.

1. Formatting
    - Whitespace: spaces are used as delimiters – separate tokens of the language. Extra spaces (horizontal, vertical) are irrelevant.
    - The language is case-sensitive.
    - The extension of the source code files is *iol* (e.g. *test.iol*).

2. Coding: A valid IOL code starts with the word ***IOL***, followed by all the "program statements," and ends with the word ***LOI***.
    - Before the word ***IOL***, there should only be whitespace or the start of the file.
    - After the word ***LOI***, there should only be whitespace or the end of the file.
    - Between ***IOL*** and ***LOI*** are the "statements" of the program code. These "statements" are the expressions and operations.

3. Data types: each variable can be defined using one of the following data types:
    - Integer type (***INT***): this is the only numeric type. An integer literal is described by the following EBNF (ignore the spaces):

        int → *digit* { *digit* }

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- String type (***STR***): there is no literal for this type. The value of the variable can only be user-input.

4. Variable: storage for a value. A variable is a simple expression that evaluates to its corresponding value.
   - Naming a variable follows the format:
     var_name → *letter* { (*letter* | *digit*) }
     letter → a | b | c | … | z | A | B | C | … | Z
     digit → 0 | 1 | 2 | … | 9
   - To use a variable, it must first be defined. Defining a variable takes either of the forms:
     data_type var_name
     or
     data_type var_name ***IS*** *value*
   - A variable must be defined before it can be used. For a variable that is defined with no initial value, its initial value will be the default value (0 for numeric type, empty string for string type).

5. Assignment, input, and output operations:
   - In an assignment operation, *expr* should evaluate to a value that is valid based on the data type of the target variable (*var_name*).
   - An assignment operation takes the form:
     INTO *var_name* ***IS*** *expr*
   - In an input operation, only a user-input value that is valid based on the data type of the target variable (*var_name*) will be stored to the target variable.
   - An input operation takes the form:
     BEG *var_name*
   - An output operation takes the form:
     PRINT *expr*
   - In the output operation form, *expr* can only be one of the following: literal, variable, numerical expression.

6. Numerical expressions: the language uses prefix notation and C's precedence and associativity rules for evaluating the expressions.
   - Simple expression: an expression that is only either a numeric literal or a numeric variable.
   - Complex expression: an expression that is a result of combining two expressions using a numerical operation. Each expression, *expr1* and *expr2*, can be either a simple expression or a complex expression.

Operation	Complex Expression	Equivalent in C
Addition (***ADD***)	***ADD*** *expr1 expr2*	*expr1 + expr2*
Subtraction (***SUB***)	***SUB*** *expr1 expr2*	*expr1 - expr2*
Multiplication (***MULT***)	***MULT*** *expr1 expr2*	*expr1 * expr2*
Division (***DIV***)	***DIV*** *expr1 expr2*	*expr1 / expr2*
Modulus (***MOD***)	***MOD*** *expr1 expr2*	*expr1 % expr2*

   - Each expression evaluates to a numeric value.

7. Built-in commands:
   NEWLN – appends a new line command (***n*** in C/C++/Java) at the end of the current output console line

8. Sample code and program run:
   - Code (*test.iol*)

```
IOL
        INT num IS 0 INT res IS 0
        STR msg1 STR msg2 STR msg3
        BEG msg1 BEG msg2
        BEG msg3
        NEWLN PRINT msg1
        NEWLN
        INTO res IS MULT num num
        PRINT msg2
        PRINT MULT num 2
        NEWLN
        PRINT msg3
        PRINT res
LOI
```

   - Compile message and Program run

```
test.iol compiled with no errors found. Program test
will now be executed…


IOL Execution:

Input for msg1: Hello, world!
Input for msg2: When doubled, the result is:
Input for msg3: When squared, the result is:

Hello, world!

Input for num: 3

When doubled, the result is: 6
When squared, the result is: 9

Program terminated successfully…
```

     Note: for the implementation, you may use a pop-up dialog (JOptionPane in Java or similar feature in the other languages) to implement the BEG command of the language to get the user input

THE IMPLEMENTATION

Note: READ AND UNDERSTAND THE FOLLOWING TEXTS CAREFULLY!!! AVOID ASKING QUESTIONS THAT CAN BE ANSWERED BY THIS DOCUMENT!

Note: Only the standard libraries of the programming language of choice (C++, Java, or Python) are allowed. Make sure that there will be no further configurations needed when your programs will be compiled and executed for checking.

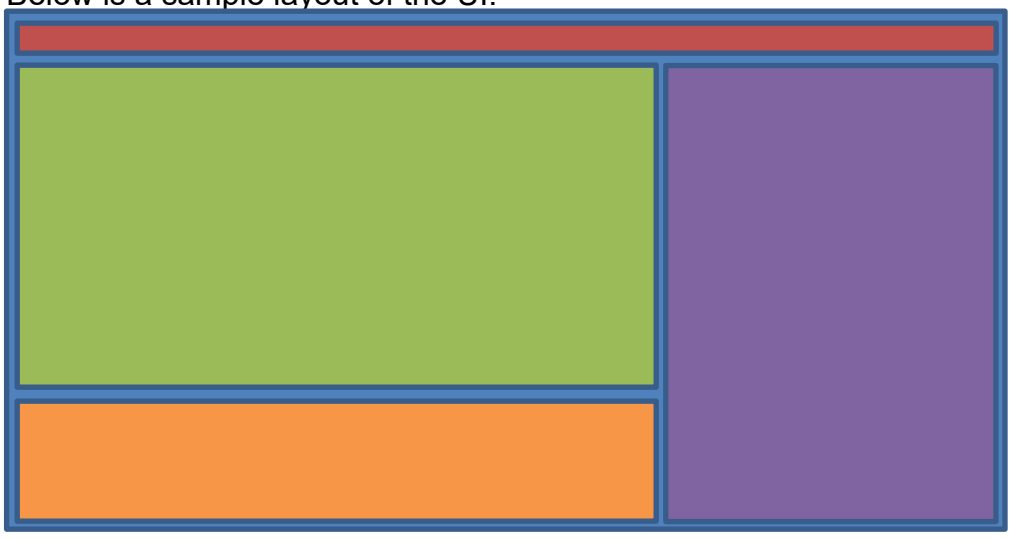
The IOL implementation will have two main components:

- Program Compilation:
  - Lexical Analysis
  - Syntax Analysis and Static Semantics Analysis

- Program Execution – actual execution of the program (based from the code) and checking of runtime correctness (type checking/runtime semantics).

Program Components:

- Program Interface (IDE; **should be very easy to use**): Using the interface, a user should be able to do the ff.:
  - Create a new source code file or open an existing source code file (***.iol*** extension) and display its content in the editor area
  - Display the name of the source code file being edited – name as the tab label of the source code area
  - Save source code in the editor area in a file (***.iol*** extension)
    - Save: save to the same file (for opened file) or to a new file (using a default filename for code create via create new)
    - Save As: save to a new file using a filename specified by the user (for both opened file and new source code)
  - Code/Edit source code in the editor area (editable text area)
  - Compile (menu button) the source code contained in the editor area
    - Execute lexical analysis: output file containing the stream of tokens, display of errors found
    - Execute syntax analysis and analysis of static semantics: display of errors found
  - Execute (menu button) the compiled source code contained in the editor area
  - Be able to read result of compilation – compile and run checking results (display only)
  - Interact with the FLOPOL program through a built-in console (runtime display)
  - Note: *Both the compilation display and the built-in console can be the same display*
  - Below is a sample layout of the UI:



        - **Menu: *Save*, *Save As*, *New File*, *Open File*, etc.**
        - **Menu: Compile Code, Show Tokenized Code**
        - **Menu: Execute Code**
        - **Built-in Code Editor**
        - **Built-in Console (status, compile and runtime execution display)**
        - **Table of variables used**
        - The compile, show tokenized code, and execute menu items should each have its own keyboard shortcut.
        - The name of the IDE should be shown on the header/title area of the frame.

- Lexical Analysis
    - Input: source code contained in the editor area
    - Scan and tokenize the input source code.
    - For tokenization (identifying which lexemes belong to which token):
        - The token name for an integer literal is **INT_LIT**
        - The token name for a keyword is the keyword itself (e.g. token name for DEFINE is DEFINE, for INTO is INTO, for IS is IS, etc.)
        - The token name for an identifier is **IDENT**
        - The token name for an error lexeme is **ERR_LEX**
    - Outputs:
        - File for the resulting stream of tokens (tokenized version of the source code). The extension for this file is ***tkn***. A stream of tokens for a source code is one in which each of the lexemes from the code is replaced by the corresponding token name.
        - Display of successful message or message for lexical error(s) found. Display should use the built-in console of the interface. Error display message should provide information about what words are unknown and where they are located (code line number).
        - A list of variables (name and type) should be displayed as a table in the area for the Table of variables used.

- Syntax Analysis and Analysis of Static Semantics
    - *Note: This follows immediately after lexical analysis*
    - Input: generated token stream file from the lexical analysis
    - Syntax analysis: parsing using either Recursive Descent Parsing or Non-Recursive Predictive Parsing or whatever method you use
    - Analysis of static semantics: type checking of expressions of the code
    - Output is result of parsing (syntax and static semantics) – display successful for analysis without problems found or unsuccessful and list of errors for analysis with problems found. Error display message should provide information about what the errors are and where they are located (code line number).

- Program Code Execution
    - *Note: This should work only after the source code contained in the editor area has been compiled successfully without any error*

- Execute the code contained in the editor area. The necessary displays (check the last part of the IOL specifications file) should be reflected on the console area of the IDE.
- Runtime type checking should be performed whenever the BEG command is executed. Once an error (mismatch between the nature of the user input and the destination variable of BEG) is encountered during this runtime type checking, the execution should terminate with an error display in the console area.

Sample code and program run:

- Code (*test.iol*)

```
IOL
        INT num IS 0 INT res IS 0
        STR msg1 STR msg2 STR msg3
        BEG msg1 BEG msg2
        BEG msg3
        NEWLN PRINT msg1
        NEWLN
        INTO res IS MULT num num
        PRINT msg2
        PRINT MULT num 2
        NEWLN
        PRINT msg3
        PRINT res
LOI
```

- Compile message and Program run

```
test.iol compiled with no errors found. Program test
will now be executed…


IOL Execution:

Input for msg1: Hello, world!
Input for msg2: When doubled, the result is:
Input for msg3: When squared, the result is:

Hello, world!

Input for num: 3

When doubled, the result is: 6
When squared, the result is: 9

Program terminated successfully…
```

Note: For the implementation, you may use a pop-up dialog (JOptionPane in Java or similar) to implement the BEG command of the language to get the user input

Programming Errors: Errors violate the language specifications. Violation on the following will mean an error in the program written using IOL:

- Lexical – unknown words found in the program code

- Syntax – not following the correct ordering of the tokens based on the valid forms specified by the language description. In this example, the BEG command is followed by the command for an operation:

  BEG ADD num num

- Type compatibility – mismatch of the types (of operands, variables, etc.) with respect to the operations. In this example, *str* is a string-type variable.

  ADD 4 str

  Possible error message: Incompatible types found for the operation ADD: expected two numerical expressions but one given is not [str]

SUBMISSION REQUIREMENT

Your submission should contain the following archived in a zip file:

- Project Folder containing the source code files of your program
- "Executable" file (e.g., JAR file for Java programs) of the program
- Documentation file containing the design details of your implementation, information on task distribution (who did what), user manual, etc. This should be a formal document.
- A 5-minute video (screen capture) showing and explaining the main parts of your code and demonstrating the use of the IDE.

Name the zip file using your surnames (alphabetical order, separated by underscore) followed by the PE # similar to this example: Bonifacio_Rizal_PE04.