Programming Project
Implementation of a Custom Programming Language

OBJECTIVE

To be able to write a program that implements a given custom programming language.

INSTRUCTIONS

1. You are to work on this activity by group (4 members).
2. You are to write a code for a program that implements a given custom programming language. 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. Use C/C++, Java, or Python in writing the code of your program. Only the standard libraries of the programming language of choice (C/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.
4. All your program files (source code files, input 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. You only need to submit the source code file you have created. If you created more than one source code file (i.e. source code is broken down into several files) or there are other files (e.g. input files) involved, archive them in a single zip file. Name your source code file or zip file, whichever is applicable, using your surname similar to this example: Bonifacio_Minda_Rizal_Project (names to be ordered alphabetically).

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

PROGRAM SPECIFICATIONS

The name of the custom language is Simple Number-Only Language (SNOL). SNOL is a simplified custom language that only involves integer and real values, operations, and expressions.

The following set of specifications covers the rules that should be observed and implemented in using SNOL:

1. Formatting
    - Whitespace
        - Spaces are used to separate tokens (constructs) of the language but are not necessary. A command can have no space in it.
        - Extra spaces are irrelevant
    - SNOL is case-sensitive
2. Data type
    - There is no type declaration
    - The type of a variable is determined by the nature of the value it holds
    - There are only two types of values: integer and floating-point
    - Writing an integer literal follows the format described by this EBNF rule:
      [-]*digit*{*digit*} where: *digit* represents a number from 0 to 9
    - Writing a floating-point literal follows the format described by this EBNF rule:
      [-]*digit*{*digit*}.{*digit*} where: *digit* represents a number from 0 to 9
3. Variable
    - A variable is considered as a simple expression that evaluates to its corresponding value
    - Naming a variable follows the format described by this EBNF rule:
      letter{(*letter*|*digit*)}
      where: *letter* represents a symbol from a to z and from A to Z
      where: *digit* represents a number from 0 to 9
      note: a keyword cannot be used as a variable name

- **Variables must be defined first** before they can be used
- To define a variable, its first use must be one of the following:
    - as the destination variable in an assignment operation
    - as the target variable in an input operation
- A variable that has already been defined to be of a certain type can be redefined to be of a different type by subjecting the variable to either the assignment operation (as the destination variable) or the input operation (as the target variable).

  Example: num = 0 Explanation: this defines the variable num to be of type integer since its value is 0 (an integer)

  BEG num Explanation: if a floating-point value is given here, num will become a floating-point type variable

4. Assignment, input, and output operations
    - Assignment operation:

      var = *expr*

      where: *var* represents the destination variable (where the value of *expr* will be stored)

      where: *expr* is an arithmetic operation, a variable name, or a literal
    - Input operation:

      BEG *var*

      where: *var* is the target variable (where the user-input value will be stored)
    - The input operation is similar to C's *scanf* function and is for asking user-input value
    - Output operation:

      PRINT *out*

      where: *out* is either a variable name or a literal
    - The output operation is similar to C's *printf* function and is for displaying value
    - Each of the assignment, input, and output operations does not evaluate to any value when executed.
5. Arithmetic operations
    - Each operation is in infix notation and follows C's precedence and associativity rules

- Operations:

Addition	*expr1* + *expr2*	Same as: *expr1* + *expr2* in C
Subtraction	*expr1* - *expr2*	Same as: *expr1* - *expr2* in C
Multiplication	*expr1* * *expr2*	Same as: *expr1* * *expr2* in C
Division	*expr1* / *expr2*	Same as: *expr1* / *expr2* in C
Modulo	*expr1* % *expr2*	Same as: *expr1* % *expr2* in C

  where: *expr1* is either a literal or a variable

  where: *expr2* is either a literal or a variable

  For the modulo operation, only integer type is allowed for both *expr1* and *expr2*

- Each operation is an expression that evaluates to its corresponding result

  Example: the command “1 + 2” evaluates to 3

- **All operands in an operation should be of the same type**

  Example: the command “2 + 1.5” is not valid since the operands 2 and 1.5 are not of the same type

6. Exit operation: When given, this operation terminates the interpreter environment. The exit operation takes the form: ***EXIT!***
7. Command: A command can be any valid literal, variable, and operation.

Sample run of the program you will be writing to implement SNOL and the corresponding explanation of each part:

Sample run:	*Explanation:*
The SNOL environment is now active, you may proceed with giving your commands.	>> Introductory message. Displayed at the start of the interpreter run.
Command: num = 0	>> Assignment operation on [num]
Command: PRINT num SNOL> [num] = 0	>> Output operation on [num]
Command: BEG var SNOL> Please enter value for [var] Input: 75	>> Input operation on [var] >> Interpreter is asking for the value to be stored to [var]
Command: 1 + var2 SNOL> Error! [var2] is not defined!	>> trying to add 1 and the value of [var2], but [var2] is not defined
Command: num + 1.5 SNOL> Error! Operands must be of the same type in an arithmetic operation!	>> trying to add the value of num and 1.5, but the two values are not of the same type
Command: BEG num SNOL> Please enter value for [num]: Input: 25.3	>> Input operation on [num]. By giving it a floating-point value, its type now becomes floating-point.
Command: num + 1.5	>> Adding the value of [num] and 1.5
Command: + num 6 SNOL> Unknown command! Does not match any valid command of the language.	>> trying to give a command that is not recognized by the language.
Command: PRINT num SNOL> [num] = 25.3	>> Display value of [num]
Command: num + BEG num 3 SNOL> Unknown command! Does not match any valid command of the language.	>> trying to give a command that is not recognized by the language.
Command: num = 2 + 3	>> Assignment operation on [num] with the value coming from the arithmetic operation involved.
Command: PRINT num SNOL> [num] = 5	>> Print operation on [num]
Command: var	>> Simply specifying a simple expression.
Command: EXIT!	>> Invoking the exit command
Interpreter is now terminated…	>> Program execution is terminated

For the implementation:

1. In the sample run, the lines that start with the word “Command: “ are the command lines. Whenever the interpreter is in execution: the interpreter asks the user for a command, the user gives the command, then the interpreter executes the command. This is repeated until the command ***EXIT!*** is executed.
2. Only one command at a time is allowed to be given in the command line. When the interpreter asks for a command, the user is to type the command then press enter.
3. In the sample run, the lines that start with “Input: “ are for when the interpreter is asking for a value to be stored to the specified variable. This only appears when the command given is ***BEG***. When the interpreter asks for an input, the user is to type the input then press enter. Everything the user types before pressing the enter key should be treated as the input the user is trying to give.
4. Command execution:
    - When BEG is encountered: display message similar in the sample run and ask the user for an input value and store it to the target variable. No message display after command is successfully executed.
    - When PRINT is encountered: display the value of the operand.
    - When an arithmetic operation is encountered, perform the corresponding operation. No message display when command is successfully executed.
    - When the input operation is encountered, evaluate the operand and store its value to the destination variable. No message display when command is successfully executed.
5. Checking for errors:
    - Unknown word: any word that is not a keyword (e.g., ***BEG***, ***EXIT!***, etc.), not a valid literal, and not a valid variable name.
      Sample command: !num = 0
      Sample message: Unknown word [!num]
    - Undefined word: a valid variable name that has not been defined (associated to a type).
      Sample command: var + 2 // Assume that this is the first use of *var*
      Sample message: Undefined variable [var]
    - Unknown command: a command that does not completely follow the correct syntax specified in the language.
      Sample command: BEG 5

Sample command: num + 2 5

- Incompatible types: encountered when an arithmetic operation has operands that are not of the same type
  Sample command: 2 + 1.5
- Invalid number format: encountered when the user attempts to enter an input that contains symbols not related to the format of either integer or floating-point. Since the only data types allowed are integer and floating-point, the interpreter should check if the input given by the user follows either integer or floating-point literal format.

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.

PROJECT SUBMISSION

What are to be included in your project submission? The following archived in a zip file named using your surnames (in alphabetical order; use underscore symbol as separator of two adjacent surnames):

- Project Folder containing the source code file(s) of your program
- "Executable" file of your program
- Documentation file containing the design details of your implementation (breakdown into classes/objects, functions, etc.), information on task distribution (who did what), user manual (how to use the program), etc. This should be a formal document (Font: TNR 12, Margin: 1" all sides, Spacing: 1.5).
- A 5-minute video (screen capture; suggestion: use Zoom) demonstrating the use of your interpreter program.

COMPLETION REQUIREMENTS FOR THE PROJECT

- Project submission: deadline is the last day of classes for the semester
- Project presentation and demonstration: during the final exam week