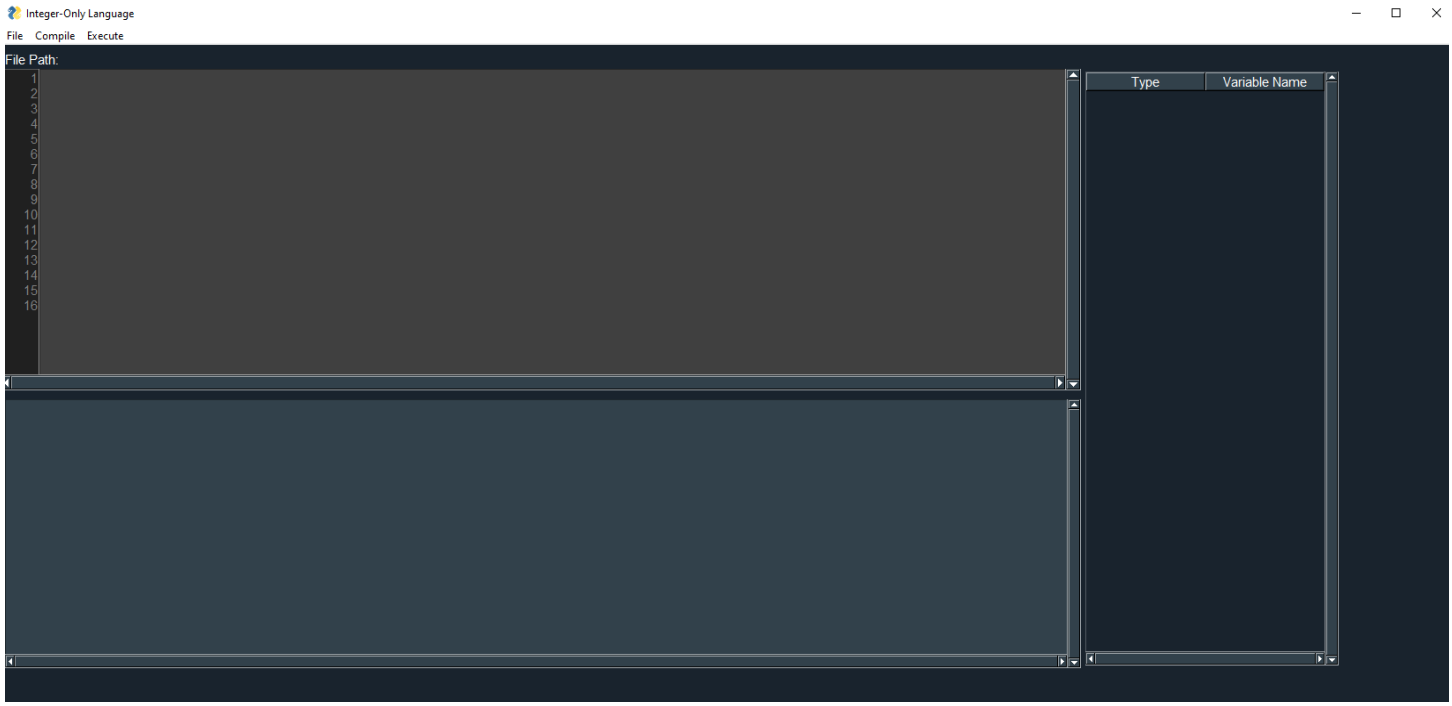
Programming Project: Integer-Only Language (IOL)

Program Documentation

UI and Design of the Program



A. Description of the Program

Python was used to create the program. The PySimpleGUI library was used to create the design and UI of the program. An event loop will be used to capture any event or action that has been done within the window. A series of *elif* statements from this event loop will determine what lines of code should run depending on what button in the submenu the user clicks on within the UI.

The user must open an input file with the extension '*.iol*,' or type out the code in the editor area and use the *Save As* function. The *Open File* under the *File* menu is used to open an input file. After loading the input file, the user must click *Compile Code* to start analyzing the code. When the user clicks on *Compile Code*, lexical and syntax analysis starts. Errors, and the line numbers they are found in, will be

displayed on the console afterwards. If there are no errors found during lexical and syntax analysis, semantic analysis will begin. Errors, and the line numbers they are found in, will be displayed on the console.

The user should *Compile Code* before clicking on *Show Tokenized Code* or *Execute Code*.

If the content of the current file is empty, then the program will not compile the input. The program will display an error window instead.

B. Parts of the program

Files

main.py

This file contains all of the code for the user interface. The details of the file are described in the next sections of this documentation file.

dynamic_multiline.py

dynamic_multiline.py is a python file that is used to show line numbers beside the editor area. It contains a class with functions to align and display the lined numbers properly.

Reference: https://github.com/PySimpleGUI/PySimpleGUI/issues/5934

lexical_analyzer.py

lexical_analyzer.py contains the code for the implementation of the lexical analysis during compilation. The function of this module will be discussed in the “Lexical analyzer” section of this document.

syntax_analyzer.py

syntax_analyzer.py contains the code for the implementation of the syntax analysis during compilation. The function of this module will be discussed in the “Syntax/Semantic analyzer” section of this document.

semantic_analyzer.py

lexical_analyzer.py contains the code for the implementation of the semantic analysis during execution. The function of this module will be discussed in the “Syntax/Semantic analyzer” section of this document.

expression_evaluation.py

expression_evaluation.py contains a class with various methods used for the evaluation of numerical expressions. The function of this module will be discussed in the “Program Code Execution” section of this document.

Parts and Function of the Program

Frontend

PySimpleGUI library was used to create and design the GUI of the program; the code for this will form the graphic components of the program.

The program will display the editor console on the upper left of the screen, an output console on the lower left, and a variable table on the rightmost.

The program will also have a menu bar on the upper left corner of the dialog box, consisting of the following tabs:

- File tab

    - *Save* - save current file from editor console
    - *Save As* - save inputs from editor console to another file
    - *New File* - clears editor console
    - *Open File* - opens file
    - *Exit* - exits program
- Compile tab
    - *Compile Code* - execute lexical, syntax, and semantic analysis from input file
    - *Show Tokenized Code* - shows tokenized code from input file and create a *output.tkn* file where the output code is written

- Compile tab
    - *Compile Code* - execute lexical, syntax, and semantic analysis from input file

Lexical analyzer

When *Compile Code* is chosen, the input file will then be read and have each line be stored in a list, checking each character and label into its appropriate token names, namely:

- INT_LIT - numerical integers
- IDENT - identifiers, specifically variables
- Keywords such as BEG, INTO, IS, PRINT, and NEWLN (token name is the keyword itself)
- ERR_LEX - error lexeme (unidentifiable character)

The output tokens determined from input are then stored in another list. Tokens encoded as 'INT' and 'STR' corresponding to data types and their corresponding variable names will be displayed in a table where the data type and its variable are paired.

All of these tokens are then sent to the next stage, which is the syntax/semantic analyzer module.

Syntax/Semantic analyzer

Syntax analysis occurs after the generation of tokens by the lexical analyzer. This segment of code checks every single string in each line from the .iol file. It checks if certain groups of keywords are compatible with the grammar of the language. If the syntax analysis is successful without any errors, then semantic analysis will proceed. In semantic analysis, the tokens are compared with keywords and data types to catch semantic errors.

Syntax analysis includes the following components:

- IOL/LOI
    - Checks if the IOL and LOI keywords are not at the start and end, respectively. If not, an error will be thrown and analysis will not execute
- Data Types
    - Checks if the declared data type has an identifier.
- Variable/Identifier
    - Checks *INT var_name IS value* statement is valid.
    - Checks if the naming of the identifier is valid
- Assignment, input, and output operations
    - Checks if *INTO var_name IS expr* is valid, and if the identifier is numeric type.
    - Checks if BEG is using an identifier
    - Checks if PRINT is using literal, variable, or numerical expression
- Numerical Expressions

    - Checks if numeric literal and numeric variables were used
    - If a complex expression was found, check if the operations result in a numeric value.

Semantic analysis involves the following components:

- Assignment operation
    - Checks if *INTO var_name IS expr* statement has variable as the simple expression. If it is a variable, then check if it is a numeric type
- Variable
    - Checks if the variable being defined with a numeric value is of numeric type
- Numerical Expressions
    - Checks if complex expressions involve variables with numeric type

Program Code Execution

When a *.iol* file is opened, it forces the user to first compile the code. Execution of the code occurs only if the source code contained in the editor area has been compiled successfully without any error.

Program execution involves the reflection of necessary displays on the console. Runtime type checking is done whenever the BEG command is executed. Execution involves the following:

- Once a mismatch between the nature of the user input and the destination variable of BEG is encountered, execution terminates with an error display in the console area.
- Modulo by zero and division halts program execution, and a pop-up window shows the error.

- The user must input a valid string for a string type variable. Likewise, a positive integer must be inputted for the integer type variable. Invalid inputs, or an empty input for an integer type variable, immediately halts execution.

User-Defined Functions in *main.py*

delete_file()

Function for deleting *console.txt* and *execute.txt* after program exit

get_file_name()

function for acquiring file name; returns file name from its path

save_file_as()

function for saving another file; opens dialog box;

matching_keyword()

gets two arguments to compare keywords on a list and determine similarity; returns bool value

remove_sublist()

removes a sublist within a list; returns the remaining of that list

Methods in class from *expression_evaluation.py*

checkIfInteger()

Method to check if the lexeme is an integer

checkIfVariable()

Method to check if the lexeme is a variable

evaluateVariableValue()

Method to get the integer value of a given variable

expressionSyntaxEvaluation()

Method for expression evaluation; checks if expression follows correct syntax

expressionEvaluation()

Method for evaluating numerical expression into a numeric value

C. Control flow of the program

Highlighted in **bold** are UI control elements, highlighted in **bold** are main operations

File

New file (Ctrl + N)

Clears content of current file

Open file (Ctrl + O)

Open and display IOL File contents

Save (Ctrl + S)

Saves current code in editor to file.

Save as (Ctrl + Shift + S)

Saves code in editor to a new file

Compile

Show Tokenized Code (Ctrl + Alt + T)

If not compiled yet, display a pop-up window telling the user to compile code first.

Otherwise, display tokenized code on the output screen.

Compile Code (Ctrl + Alt + C)

When the user clicks on Compile Code, lexical, syntax, and semantic analyses occur:

- **Lexical Analysis**

  If invalid lexeme(s) is found, print list of invalid lexemes.

  Regardless of whether invalid lexemes were detected or not, display variable's names and types on the top right table, and proceed to syntax analysis.

- **Syntax Analysis**

  If no errors were detected, display a 'no errors detected' message and immediately proceed to semantic analysis

  Otherwise, throw a list of error messages, and the code lines from where these errors were found.

- **Semantic Analysis**

  If no errors were detected, display a 'no errors detected' message

  Otherwise, throw a list of error messages, and the code lines from where these errors were found

Compile

Execute code (Ctrl + Alt + E)

If any errors were detected during execution, then program execution halts immediately and a pop-up window will show the respective error.

Exit (Ctrl + X) - Close program

D. Input File Specifications

The input file must contain the file extension *.iol* in order to be properly read. The *.iol* must have IOL as the first line of the file, and must have LOI as the last line of the file. This will help the program identify where the start and end of the code. Spaces are used as delimiters and inputs are case-sensitive.

1. For data types, a literal integer is described by the following EBNF:

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

   For strings, the value of the variable must be only user-defined through an input, which follows the following EBNF:

   var_name → letter { (letter | digit) }
   letter → a | b | c | ... | z | A | B | C | ... | Z
   digit → 0 | 1 | 2 | ... | 9

   Variables can only be used after it is first defined. This can be done by:

   data_type var_name ; or
   data_type var_name IS value

2. Assignment of values into a variable must be of the same data type, and is done through:

   INTO var_name IS expr

   With *expr* being a valid evaluated value for the variable.

   Inputting values into a variable is done through:

   BEG var_name

   Outputting variable values and expressions is done through:

PRINT expr

3. Numerical expressions are done with a prefix notation through the following keywords:

    a. Addition (*ADD*)
    b. Subtraction (*SUB*)
    c. Multiplication (*MULT*)
    d. Division (*DIV*)
    e. Modulus (*MOD*)

    These operations can be done with either INT-defined variables or standard integers.

4. NEWLN performs a new line append to the current console line.

E. Work Distribution

1. Warain - in charge of the main code and algorithm dealing with lexical, syntax, and semantic analysis; created program execution of the program; integrated backend to the frontend; dealt with error-checking and documentation; created the frontend and UI of the program
2. Suico - in charge of modularization of code and documentation; assisted Chester with the algorithm for lexical analysis; error-checking
3. Chester - in charge of documentation; created the algorithm and code for lexical analysis; assisted with code for program execution;
4. Lumuntod - created the file saving and file opening functionalities of the program; in charge of documentation and error-checking