Members:

Cagay, Rod Jhon

Durias, Cedrick Jared

Tanjay, Camyl Magdalyn

I. PROGRAM DESCRIPTION

This program was developed using the Python programming language and serves as a non-recursive predictive parser for conducting syntax and semantic analysis on a custom programming language referred to as IOL (Integer Oriented Language). The application provides functionality for reading, displaying, and editing the contents of uploaded files with a (.iol) extension within an integrated editor. It facilitates the generation and export of tokenized code to an external file with a (.tkn) extension. The graphical user interface (GUI) includes features such as status updates during compilation and parsing processes, offering a comprehensive environment for IOL programming. The main buttons in the GUI are categorized for file handling (new, open, save, save as, and close) and compilation processes (compile and show tokenization), enhancing user interaction and ease of use.

II. PARTS/MODULES/FUNCTION OF THE PROGRAM

This is a Python program that creates a simple GUI (Graphical User Interface) application for a Programming Language Compiler using the `tkinter` library. This program is a Tkinter-based graphical user interface (GUI) application written in Python. The program includes a basic text editor for writing code, a menu bar for file handling operations, a compile button, a button to show tokenized code, an execute button, and a table to display variables and their types. Below are the main parts/modules/functions of the program:

. Below are the main parts/modules/functions/classes of the program:

Classes

- **LexicalAnalyzer -** the LexicalAnalyzer class is responsible for performing lexical analysis on the provided code in the IOL programming language. It identifies keywords, data types, and variables, generating a list of tokens in the process.

Main Functions

1. **__init__(self)**: Initializes the LexicalAnalyzer object.

   Attributes:

   - `***keywords***`: Tuple of reserved keywords in the IOL programming language.
   - `***datatypes***`: Tuple of data types in the IOL programming language.
   - `***tokens***`: List to store the tokenized representation of the code.
   - `***variables***`: Set to store unique variables and their corresponding data types.
2. **analyze(self, code)**: Performs lexical analysis on the provided code, extracting tokens and identifying variables.
   - `***lines***`: List of lines obtained by splitting the code.
   - `***token_list***`: List to store tokens for each word in a line.
   - `***var_presence***`: List to check the presence of a variable in the set of variables.
3. **get_token(self, word**): Determines the token for a given word based on lexical rules. This returns a token corresponding to the given word.
   - `***self.keywords***`: Tuple of reserved keywords.
   - `***self.datatypes***`: Tuple of data types.
4. **show_errlex(self)**: Generates a string containing error messages for unknown words detected during lexical analysis. This returns a string containing error messages for lexical analysis.
   - `***self.tokens***`: List of tokens generated during lexical analysis.

- **SyntaxAnalyzer** - The SyntaxAnalyzer class is responsible for analyzing the syntax of the provided list of tokens using the parsing table. The implementation used is the Non-Recursive Predictive Parsing which refers to a predictive parsing technique where recursive calls are avoided, and a stack is used to keep track of the parsing state.

  Main Functions

  1. `**__init__(self)**`: Initializes the SyntaxAnalyzer object with the production rules (iol_prod) and parsing table (iol_ptbl) for the IOL programming language.
  2. `**analyze(self, tokens, lexical_analyzer)**`: Performs syntax analysis on the provided list of tokens using the parsing table. It collaborates with a `***LexicalAnalyzer***` object (`**lexical_analyzer**`) to integrate lexical and syntax analysis seamlessly.
     - `***input_buffer***`: List representing the input buffer of tokens.

- `***stack***`: List representing the stack during parsing.
- `***current_production_id***`: Integer representing the current production rule being applied.

Important Production Rules (iol_prod)

1. `**"s", "IOL stmts LOI"**`: Represents the start symbol and the overall structure of the IOL program.
2. ` **"stmts", "stmt stmts"** `: Represents a sequence of statements.
3. ` **"stmts", "e"** `: Represents an empty production, indicating the end of statements.
4. ` **"stmt", "var", "stmt", "asn", "stmt", "out", "stmt", "expr", "stmt", "NEWLN"** `: Different types of statements in the IOL language.
5. ` **"var", "INT IDENT varend", "var", "STR IDENT varend"** `: Represents variable declarations with optional initial values.
6. ` **"varend", "IS INT_LIT", "varend", "e"** `: Represents the optional initialization part of a variable declaration.
7. ` **"asn", "INTO IDENT IS expr"** `: Represents an assignment statement.
8. ` **"out", "PRINT expr"** `: Represents an output statement.
9. ` **"expr", …** `: Represents various arithmetic expressions.

Functions (Non-Class)

1. compile_code(event=None) - compiles the code by performing lexical analysis, displaying errors, tokenizing the code, and saving the compiled code.

- `***editor_path***`: The file path associated with the editor content.
- `***new_file_created***`: Boolean indicating whether a new file has been created.
- `***tokenized_code***`: String containing the tokenized code.
- `***compiled***`: Boolean indicating the compilation status.

2. show_tokenized_code(event=None) - Shows the tokenized code in a new window if the code has been successfully compiled.

3. execute_code(event=None) - Placeholder function for executing the code.

4. display_variables(variables) - Displays the variables and their data types in the variables table.

5. update_status(message)- Updates the status bar with the provided message.

6. open_file(event=None) - Opens a PL file, reads its content, and displays it in the editor.

7. new_file(event=None) - Creates a new file with default content and updates the editor.

8. save_file(event=None) - Saves the content of the editor to the current file or prompts for a filename if not saved before.

9. save_file_as(event=None) - Prompts the user for a filename and saves the content of the editor to the specified file.

10. close_file(event=None) - Closes the current file, discarding any unsaved changes.

11. update_line_numbers(event=None) - Updates the line numbers in the line number section based on the current editor content.

- ***`linenum0`***: Integer representing the number of lines in the editor.
- ***`line_num`***: Tkinter ScrolledText widget for displaying line numbers.
- ***`editor`***: Tkinter ScrolledText widget representing the editor.

12. on_editor_scroll(*args) - Updates the line numbers when the editor is scrolled.

III. PROGRAM FLOW CONTROL

1. Once the program runs, the Tkinter GUI window is created and its components set up.
2. The user can perform various file operations, such as "New," "Open," "Save," or "Save as...under the File Menu.
3. If the input (.iol) files are loaded through an Open Button, the contents of the file will be loaded in the editor frame in the GUI.
4. If "Save" or "Save as..." is chosen, a file dialog opens, and the user specifies the file name and location.
5. The user interacts with the code editor and writes or modifies code in the editor.
6. Scroll through the code using the scrollbar in the editor.
7. The user can compile the code through "Compile Code" under compile in the menu bar. Lexical, semantic, and Non-recursive predictive parsing is performed.

   For lexical analysis;

a. If no lexical errors are found, the status is updated with "Code tokenized successfully."
b. If lexical errors are found, the status is updated with an error message.

For Non-Recursive Predictive Parsing;

c. If the given string is valid or invalid, the parsing status will be updated. For variable table display, the user can view the table of variables and their types:

8. The user can view the tokenized version of the code through "Show Tokenized Code" under "Compile" in the menu bar
9. A new window displays the tokenized code.
10. For variable table display, the user can view the table of variables and their types:
11. After successful compilation, the variables table is updated with variable information.
12. The user can execute the code through the "Execute" button in the menu bar.
13. Throughout the process, the user will give status updates and
14. Messages are updated during file operations, compilation, parsing, and execution.
15. The program terminates "Close" in the "File" menu.

IV. WORK DISTRIBUTION

Program UI/Status Display/Menu Process: Cagay/Tanjay

Reading and display of files: Cagay/Tanjay

Process: Durias/Tanjay