Faculty of Computer Science Spring Semester 2023 CS334

Compiler Design Project

|  |  |
| --- | --- |
| **Student Name** | **Student ID** |
| Steven Hany Saad | 210991 |
| Mina Samir Ghobrial | 212257 |

**Introduction**

Compiler design is a fundamental discipline that lies at the heart of modern programming languages, acting as the bridge between human-readable code and efficient machine execution. Compiler design involves transforming high-level programming languages into low-level machine code, encompassing stages such as lexical analysis, syntax parsing, semantic analysis, code optimization, and code generation. With its paramount role in enabling the translation, optimization, and error detection of programs.

**Project idea:**

The idea of our project is to build a simple compiler that checks on the syntax of Declaration, syntax of While loop and syntax of Switch cases.

Also, to make a memory for all variable we have in the code provided.

**The GUI:**

**A screen shot of a computer code

Description automatically generated with low confidence**

This code defines a graphical user interface (GUI) using the Tkinter library in Python. The GUI consists of several components:

* Coding Area Frame: It is a container frame where the coding area is placed. It is displayed at the top-left position of the GUI.
* Coding Area: It is a text widget where the user can input code. It is placed within the Coding Area Frame. The text widget has a fixed height and width and uses a specific font. It also has a custom color scheme for the background and foreground.
* Scrollbar 1: It is a vertical scrollbar associated with the Coding Area. It allows the user to scroll through the content of the Coding Area when there is more text than can be displayed.
* Output Area Frame: It is another container frame placed below the Coding Area Frame.
* Output Area: It is a text widget where the program output will be displayed. It is placed within the Output Area Frame. Similar to the Coding Area, it has a fixed height and width, a specific font, and a color scheme for the background and foreground. The state is initially set to "disabled," meaning the user cannot modify its content.
* Scrollbar 2: It is a vertical scrollbar associated with the Output Area. It allows scrolling through the program output when there is more content than can be displayed.
* Output Area2: It is another text widget placed beside the Output Area. It is intended to display additional information, such as memory-related data. It has the same properties as the Output Area, including a vertical scrollbar (Scrollbar 3).
* Run Button Frame: It is a container frame placed below the Output Area Frame.
* Run Button: It is a button widget labeled "Run." When clicked, it triggers the "Submit\_Code" method. It has a fixed width, a specific font, and a custom color scheme for the background and foreground. The button appearance changes when the mouse pointer enters or leaves its area.

Additionally, the code sets some initial content for the Output Area and Output Area2, and configures them to be initially disabled for editing. The Run Button is bound to events for changing its background color when the mouse pointer enters or leaves its area.

**Code Submission:**

**A screen shot of a computer program

Description automatically generated with medium confidence**

The Submit\_Code method is called when the user clicks on the "Run" button in the GUI. It performs several operations which we are going to explain, making it like the main.

self.User\_Input = self.Coding\_Area.get("1.0", tk.END): This line retrieves the content of the Coding Area text widget and assigns it to the self.User\_Input variable. It uses the get method with the range "1.0" to tk.END to get all the text from the first character to the end of the widget which is going to be used in the next operations.

**The Tokenization:**

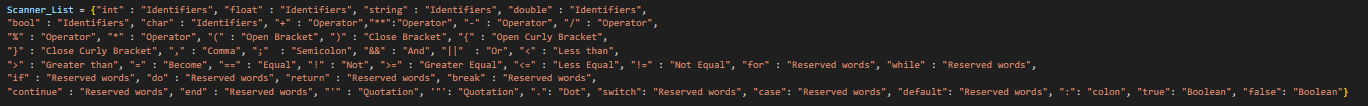
**A picture containing text, screenshot, font

Description automatically generated**

The Tokenize method is responsible for tokenizing the user's input code:

* self.Lines = self.User\_Input.strip().split('\n'): This line takes the content of self.User\_Input, which represents the user's input code, removes any leading or trailing whitespace using the strip method, and splits it into a list of lines using the newline character ('\n') as the delimiter. Each line of code is stored as an element in the self.Lines list.
* self.Tokenized\_Lines = self.Lines.copy(): This line creates a copy of the self.Lines list and assigns it to the self.Tokenized\_Lines variable. This step ensures that the original self.Lines list is not modified during the tokenization process.
* Pattern = r'([-]?\d\*\.\d+|\d+|[A-Za-z]+(?:\d+[A-Za-z]\*|\b)|\'[A-Za-z]+\'|\"(?:[^\"\\\n]|\\\"|\\\\)\*\"|[!=<>]=|\|\||&&|\\*\\*|[^\s\w\n])': This line defines a regular expression pattern that matches various types of tokens. The pattern includes patterns for numbers (both integers and floats), identifiers, string literals, comparison operators, logical operators, and other special characters.
* for Index in range(0, len(self.Lines)):: This line starts a loop that iterates over each line of code in the self.Lines list.
* self.Tokenized\_Lines[Index] = re.findall(Pattern, self.Lines[Index]): Inside the loop, re.findall is used to find all occurrences of the pattern in the current line of code (self.Lines[Index]). The resulting matches are stored as a list of tokens in the corresponding index of the self.Tokenized\_Lines list.
* self.Final\_List = self.Tokenized\_Lines.copy(): After tokenizing all the lines of code, a copy of self.Tokenized\_Lines is assigned to self.Final\_List. This step ensures that the original self.Tokenized\_Lines list is preserved and can be accessed later.
* print(self.Final\_List): This line prints the self.Final\_List to the console. It serves as a debugging or verification step to check the tokenization output.

**The Scanner List:**

****

The Scanner\_List is a dictionary that maps tokens to their corresponding types. Each key-value pair in the dictionary represents a token and its associated type:

* Keywords and Identifiers:
* "int", "float", "string", "double", "bool", "char": These tokens are classified as "Identifiers", which typically represent data types or user-defined identifiers in the code.
* Operators:
* "+", "\*\*", "-", "/", "%", "\*": These tokens are classified as "Operator". They represent mathematical and arithmetic operators.
* Brackets and Braces:
* "(" : "Open Bracket", ")" : "Close Bracket", "{" : "Open Curly Bracket", "}" : "Close Curly Bracket": These tokens represent different types of brackets and braces used in the code structure.
* Comma and Semicolon:
* "," : "Comma", ";" : "Semicolon": These tokens represent the comma and semicolon punctuation marks used in the code.
* Logical Operators:
* "&&" : "And", "||" : "Or": These tokens represent logical operators for conjunction (AND) and disjunction (OR).
* Comparison Operators:
* "<" : "Less than", ">" : "Greater than", "=" : "Become", "==" : "Equal", "!" : "Not", ">=" : "Greater Equal", "<=" : "Less Equal", "!=" : "Not Equal": These tokens represent various comparison operators used in the code.
* Reserved Words:
* "for", "while", "if", "do", "return", "break", "continue", "end", "switch", "case", "default": These tokens are classified as "Reserved words". They represent keywords that have special meanings in the programming language.
* Quotations:
* "'" : "Quotation", "\"" : "Quotation": These tokens represent single and double quotation marks used to delimit string literals.
* Dot and Colon:
* "." : "Dot", ":" : "Colon": These tokens represent the dot operator and the colon punctuation mark used in the code.
* Boolean Values:
* "true" : "Boolean", "false" : "Boolean": These tokens represent boolean literals, which can take the values true or false.

**The Scan:**

A picture containing text, screenshot

Description automatically generated

The Scan method is responsible for analyzing and categorizing the tokens in the self.Final\_List based on their types using the Scanner\_List dictionary:

* Initialize self.Scanner\_Result as a nested list with the same dimensions as self.Final\_List, filled with default values of 0 for the token and token type.
* Initialize an empty list self.Declared\_Variables to store the declared variables encountered during scanning.
* Iterate over each line in self.Final\_List:
* For each token at a specific index in the line:
* Set Flag\_Found to False to keep track of whether the token has been identified or not.
* Iterate over the key-value pairs in the Scanner\_List dictionary.
* If the current token matches a key in the Scanner\_List:
* Assign the token as the first element ([0]) of self.Scanner\_Result at the corresponding line and index.
* Assign the corresponding value from the Scanner\_List as the second element ([1]) of self.Scanner\_Result.
* Set Flag\_Found to True to indicate that the token has been identified.
* Break out of the inner loop.
* If Flag\_Found is still False after iterating over all the key-value pairs:
* Check the token against specific patterns using regular expressions to determine its type.
* If the token matches the pattern for a string literal:
* Assign the token as the first element of self.Scanner\_Result at the corresponding line and index.
* Set the type as "String".
* If the token matches the pattern for an integer literal:
* Assign the token as the first element of self.Scanner\_Result at the corresponding line and index.
* Set the type as "Integer".
* If the token matches the pattern for a float literal:
* Assign the token as the first element of self.Scanner\_Result at the corresponding line and index.
* Set the type as "Float".
* If the previous token's type is "Identifiers" or if the previous token is a comma and the first token in the line is an identifier:
* Assign the token as the first element of self.Scanner\_Result at the corresponding line and index.
* Set the type as "Variable".
* Append the token and the corresponding line's first token to self.Declared\_Variables.
* If none of the above conditions are met:
* Assign the token as the first element of self.Scanner\_Result at the corresponding line and index.
* Set the type as "Variable".
* Update self.Final\_List with the categorized tokens and their types stored in self.Scanner\_Result.
* Call the Check\_Output method, passing a copy of self.Scanner\_Result for further processing.

The Scan method effectively scans the tokens in self.Final\_List, matches them against predefined patterns and the Scanner\_List dictionary, and assigns the corresponding token types. It also identifies and tracks declared variables, storing them in self.Declared\_Variables for later use.

**The** **String Syntax:**

**A picture containing text, screenshot

Description automatically generated**

The Check\_String\_Syntax method is responsible for verifying the syntax of string literals in the self.Final\_List and identifying any errors related to unclosed quotation marks:

* Initialize an empty list self.String\_Syntax\_Check to store the checked tokens with string syntax.
* Iterate over each line in self.Final\_List:
* Initialize an empty list Temp\_Line to store the checked tokens for the current line.
* Initialize a flag variable flag as False.
* Iterate over each token at a specific index in the line:
* If flag is True, skip the current iteration and set flag to False.
* If the token's type is "Quotation":
* Check if there is a token at the next index in the line (Index + 1).
* If there is a next token, add an error message to Temp\_Line indicating that the quotation marks are not closed at the end of the current token concatenated with the next token.
* If there is no next token, add an error message to Temp\_Line indicating that the quotation marks are not closed at the end of the current line (self.Lines[Line]).
* Set flag to True to skip the next iteration.
* If the token's type is not "Quotation", add the token to Temp\_Line.
* Append Temp\_Line to self.String\_Syntax\_Check.
* Update self.Final\_List with the checked tokens and their potential error messages stored in self.String\_Syntax\_Check.
* Call the Check\_Output method, passing a copy of self.String\_Syntax\_Check for further processing.

The Check\_String\_Syntax method ensures that all string literals have properly closed quotation marks. It identifies any errors where quotation marks are not closed and adds corresponding error messages to the checked tokens in self.String\_Syntax\_Check. Finally, it updates self.Final\_List with the checked tokens, including any error messages, and calls the Check\_Output method to further process the checked tokens.

**The Declaration Syntax:**

**A computer code on a black background

Description automatically generated with low confidence**

The Check\_Declarations method is responsible for verifying variable declarations in the self.Final\_List and identifying any errors related to undeclared variables:

* Create a copy of self.Final\_List and assign it to self.Declaration\_Erros. This list will be used to store the checked tokens with declaration errors.
* Iterate over each line in self.Declaration\_Erros:
* Iterate over each token at a specific index in the line:
* If the token's type is "Variable", perform the following checks:
* Initialize a flag variable Flag\_Found as False.
* Iterate over each declared variable stored in self.Declared\_Variables:
* If the current token matches a declared variable, set Flag\_Found to True and break the loop.
* If Flag\_Found is False, modify the current token to include an error message indicating that the variable is not declared.
* Update self.Final\_List with the checked tokens and their potential error messages stored in self.Declaration\_Erros.
* Call the Check\_Output method, passing a copy of self.Declaration\_Erros for further processing.

The Check\_Declarations method ensures that all variables used in the code have been declared before their usage. It compares each "Variable" token in self.Declaration\_Erros with the declared variables stored in self.Declared\_Variables. If a match is not found, the corresponding token is modified to include an error message indicating that the variable is not declared. Finally, self.Final\_List is updated with the checked tokens and their error messages, and the Check\_Output method is called to further process the checked tokens.

**The Syntax:**

**A screen shot of a computer program

Description automatically generated with low confidence**

The Check\_Syntax method is responsible for performing various syntax checks on the tokens in self.Final\_List.

**The Semicolon Syntax:**

**A picture containing text, screenshot, font

Description automatically generated**

The Check\_Semicolon\_Syntax method checks the syntax of semicolons in the code:

* Create a copy of self.Final\_List and assign it to self.SemiColon\_Syntax\_Check.
* Iterate over each line in self.Final\_List.
* Check if the line is not empty (len(self.Final\_List[Line]) > 0).
* Check if the first token in the line is a keyword that doesn't require a semicolon after it (self.SemiColon\_Syntax\_Check[Line][0][0] in ["return", "break", "do"]) or if the first token is not a reserved word or a curly bracket (self.SemiColon\_Syntax\_Check[Line][0][1] != "Reserved words" and self.SemiColon\_Syntax\_Check[Line][0][1] not in ["Open Curly Bracket", "Close Curly Bracket"]).
* If the above condition is true, check if the last token in the line is not a semicolon (self.SemiColon\_Syntax\_Check[Line][len(self.SemiColon\_Syntax\_Check[Line]) - 1][1] != "Semicolon").
* If a semicolon is missing, append an error message to the line.
* If the last token in the line is a semicolon, perform additional checks:
* If the line has only one token, indicate that writing a semicolon alone in a line is not allowed.
* If the token before the semicolon is not a valid type or a reserved word (self.SemiColon\_Syntax\_Check[Line][len(self.SemiColon\_Syntax\_Check[Line]) - 2][1] not in ["Variable", "Integer", "String", "Float", "Boolean", "Error"]) and the first token is not "return", "break", or "do", indicate that writing that token before the semicolon is not allowed.
* Update self.Final\_List with the self.SemiColon\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with semicolon syntax checks.

The Check\_Semicolon\_Syntax method ensures that semicolons are correctly placed in the code according to the programming language's syntax rules. It detects missing semicolons and provides relevant error messages.

**The Variable Syntax:**

**A picture containing text, screenshot

Description automatically generated**

The Check\_Variable\_Syntax method is responsible for checking the syntax of variables in the code:

* Create a copy of self.Final\_List and assign it to self.Variable\_Syntax\_Check.
* Iterate over each line in self.Variable\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token is a variable or if it has the format f"Line {Line + 1} Variable {self.Variable\_Syntax\_Check[Line][Index][0]} Not Declared". These conditions ensure that the token represents a variable or an undeclared variable error message.
* If the token matches the variable format (re.match(r"^[a-zA-Z0-9]+$", self.Variable\_Syntax\_Check[Line][Index][0])), perform further checks:
* Check if the token before the variable is not a valid identifier, operator, assignment operator, bracket, logical operator, comparison operator, comma, or an error message. If it's not, indicate that writing that token before the variable is not allowed.
* Check if the token after the variable is not a valid operator, assignment operator, bracket, semicolon, logical operator, comparison operator, comma, or an error message. If it's not, indicate that writing that token after the variable is not allowed.
* If the variable doesn't match the expected format, indicate that the variable name should not contain special characters.
* Update self.Final\_List with the self.Variable\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with variable syntax checks.

The Check\_Variable\_Syntax method ensures that variables are used correctly according to the syntax rules of the programming language. It detects incorrect placement of variables in relation to surrounding tokens and identifies variable names that contain special characters.

**The Identifiers Syntax:**

**A picture containing text, screenshot, font

Description automatically generated**

The Check\_Identifier\_Syntax method is responsible for checking the syntax of identifiers in the code:

* Create a copy of self.Final\_List and assign it to self.Identifier\_Syntax\_Check.
* Iterate over each line in self.Identifier\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents an identifier (self.Identifier\_Syntax\_Check[Line][Index][1] == "Identifiers").
* If the token is an identifier, perform the following checks:
* If the identifier is the first token in the line (Index == 0), check if the token after the identifier is not a variable or an error message. If it is, indicate that you can't write anything after the identifier except variables.
* If the identifier is not the first token, indicate that you should not write anything before the identifier.
* If any errors are found, update the corresponding token with an error message (self.Identifier\_Syntax\_Check[Line][Index][0]) and set its type to "Error" (self.Identifier\_Syntax\_Check[Line][Index + 1][1] = "Error").
* Update self.Final\_List with the self.Identifier\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with identifier syntax checks.

The Check\_Identifier\_Syntax method ensures that identifiers are used correctly according to the syntax rules of the programming language. It detects incorrect placement of identifiers in relation to surrounding tokens and provides error messages when necessary.

**The Become Syntax:**

**A picture containing screenshot

Description automatically generated**

The Check\_Become\_Syntax method is responsible for checking the syntax of variable assignment using the "become" operator in the code:

* Create a copy of self.Final\_List and assign it to self.Become\_Syntax\_Check.
* Iterate over each line in self.Become\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents the "become" operator (self.Become\_Syntax\_Check[Line][Index][1] == "Become").
* If the token is the "become" operator, perform the following checks:
* Check if the token before the "become" operator is not a variable or an error. If it is, indicate that you can't write anything before the "become" operator except variables.
* Check if the token after the "become" operator is not a variable, integer, boolean, string, float, error, or open bracket, and it's not a negative sign ("-"). If it is, indicate that you can't write anything after the "become" operator except valid assignment values.
* If both the token before and after the "become" operator are variables, check if their types are compatible. If they are not, indicate that you can't assign a value of one type to a variable of a different type.
* If any errors are found, update the corresponding tokens with error messages and set their types to "Error".
* Update self.Final\_List with the self.Become\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with become syntax checks.

The Check\_Become\_Syntax method ensures that variable assignments using the "become" operator are performed correctly according to the syntax and type rules of the programming language. It detects incorrect usage of the operator, incompatible assignment types, and provides appropriate error messages.

**The Condition Syntax:**

A picture containing screenshot

Description automatically generated

The Check\_Condition\_Syntax method is responsible for checking the syntax of conditions in the code:

* Create a copy of self.Final\_List and assign it to self.Condition\_Syntax\_Check.
* Iterate over each line in self.Condition\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents a condition keyword (self.Condition\_Syntax\_Check[Line][Index][1] in ["And", "Or", "Less than", "Greater than", "Greater Equal", "Less Equal", "Not Equal", "Equal", "Not"]).
* If the token is a condition keyword, perform the following checks:
* Check if the first token in the line is either "while" or "case". If not, indicate that a condition cannot be written on this line without "while" or "case" at the start.
* If the first token is "while" or "case", check if the token before the condition keyword is not a variable, integer, boolean, string, float, or error. If it is, indicate that you can't write anything before the condition keyword except valid expressions.
* Check if the token after the condition keyword is not a variable, integer, boolean, string, float, or error, and it's not a negative sign ("-"). If it is, indicate that you can't write anything after the condition keyword except valid expressions.
* If both the token before and after the condition keyword are variables, check if their types are compatible. If they are not, indicate that you can't compare a variable of one type to a variable of a different type.
* If any errors are found, update the corresponding tokens with error messages and set their types to "Error".
* Update self.Final\_List with the self.Condition\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with condition syntax checks.

The Check\_Condition\_Syntax method ensures that conditions are written correctly according to the syntax and type rules of the programming language. It detects incorrect usage of conditions, incompatible comparison types, and provides appropriate error messages.

**The Operator Syntax:**

A picture containing text, screenshot, font

Description automatically generated

The Check\_Operator\_Syntax method checks the syntax of operators in the code:

* Create a copy of self.Final\_List and assign it to self.Operator\_Syntax\_Check.
* Iterate over each line in self.Operator\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents an operator (self.Operator\_Syntax\_Check[Line][Index][1] == "Operator").
* If the token is an operator, perform the following checks:
* Check if the token before the operator is not a variable, close bracket, integer, float, or error, and it's not a negative sign ("-"). If it is, indicate that you can't write anything before the operator except valid expressions.
* Check if the token after the operator is not a variable, open bracket, integer, float, or error. If it is, indicate that you can't write anything after the operator except valid expressions.
* If any errors are found, update the corresponding tokens with error messages and set their types to "Error".
* Update self.Final\_List with the self.Operator\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with operator syntax checks.

The Check\_Operator\_Syntax method ensures that operators are used correctly according to the syntax rules of the programming language. It detects incorrect usage of operators and provides appropriate error messages.

**The Integer Syntax:**

**A picture containing screenshot, text

Description automatically generated**

The Check\_Integer\_Syntax method is responsible for checking the syntax of integer and float values in the code:

* Create a copy of self.Final\_List and assign it to self.Integer\_Syntax\_Check.
* Iterate over each line in self.Integer\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents an integer or float (self.Integer\_Syntax\_Check[Line][Index][1] in ["Integer", "Float"]).
* If the token is an integer or float, perform the following checks:
* Check if the token before the integer or float is not an operator, open bracket, "Become" keyword, logical operator (e.g., "And", "Or"), comparison operator (e.g., "Less than", "Greater than"), assignment operator ("Equal"), "Not" keyword, error, or "case" keyword (for the case statement). If it is, indicate that you can't write anything before the integer or float except valid expressions.
* Check if the token after the integer or float is not an operator, close bracket, semicolon, comma, logical operator, comparison operator, assignment operator, "Not" keyword, colon, or error. If it is, indicate that you can't write anything after the integer or float except valid expressions.
* If any errors are found, update the corresponding tokens with error messages and set their types to "Error".
* Update self.Final\_List with the self.Integer\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with integer syntax checks.

The Check\_Integer\_Syntax method ensures that integer and float values are used correctly in the code. It detects incorrect placements of integers or floats and provides appropriate error messages.

**The String Advanced Syntax:**

**A picture containing text, screenshot, font

Description automatically generated**

The difference between this function and the one before it (Check\_String\_Syntax) is that this function checks the surroundings of the string while the first one checks on the quotations and how the string is written.

* Create a copy of self.Final\_List and assign it to self.String\_Syntax\_Check.
* Iterate over each line in self.String\_Syntax\_Check.
* Iterate over each token in the current line.
* Check if the token represents a string (self.String\_Syntax\_Check[Line][Index][1] == "String") and ensure it is not the first token in the line (i.e., Index != 0).
* If the token is a string, perform the following checks:
* Check if the token before the string is not an open bracket, "Become" keyword, or error. If it is, indicate that you can't write anything before the string except valid expressions.
* Check if the token after the string is not a close bracket, semicolon, comma, or error. If it is, indicate that you can't write anything after the string except valid expressions.
* If any errors are found, update the corresponding tokens with error messages and set their types to "Error".
* Update self.Final\_List with the self.String\_Syntax\_Check copy.
* Call the Check\_Output method to display the updated list with string syntax checks.

The Advanced\_Check\_String\_Syntax method ensures that string values are used correctly in the code. It detects incorrect placements of strings and provides appropriate error messages.

**The while syntax:**

**A picture containing text, screenshot

Description automatically generated**

**In this code:**

* The code initializes a copy of "Final\_List" as "While\_Syntax\_Check" and sets a counter and flag variable to zero and False, respectively. An "order\_list" is defined to represent the expected order of tokens in a while loop. The code then iterates through each token in "While\_Syntax\_Check" and compares it with the corresponding token in "order\_list." If the tokens match and are in the expected order, the counter is incremented, and the flag is set to True. If any token does not match the expected order, the counter is not incremented. After the loop, if the counter does not match the length of "order\_list" and the flag is True, an error message is appended to the last item in "While\_Syntax\_Check." Finally, the "Final\_List" is updated with the modified "While\_Syntax\_Check," and the "Output" method is called with "While\_Syntax\_Check" as an argument.

**The switch syntax:**

A screen shot of a computer program

Description automatically generated with low confidence

**In this code:**

* various counters and flags required for tracking specific tokens and their occurrences. The "order\_list" is defined to specify the expected order of tokens within a switch statement.
* iterate through each token in "Final\_List" using nested loops. For each token encountered, it compares it with the corresponding token in "order\_list" and increments the counter if there is a match. Additionally, the code checks for specific tokens such as 'switch,' 'case,' ':,' 'break,' and 'default' and increments their respective counters.
* After the loop, the code performs additional checks to ensure the correct structure of the switch statement. It examines the positions of 'default' and 'case' tokens within "Final\_List" to determine if there are any misplaced or missing case statements after the default. If any errors are found, appropriate error messages are appended to the last item in "Final\_List."
* The code checks for conditions where the switch statement structure is not correct. This includes scenarios where the number of 'case' tokens does not match the number of 'break' tokens, where the number of 'case' tokens does not match the number of ':' tokens minus one, or where there is a missing 'switch' statement altogether. In such cases, error messages are appended to the last item in "Final\_List.".
* Finally, the modified "Final\_List" is passed as an argument to the "Output" method, which presumably handles the display.

**The History:**

A picture containing text, screenshot

Description automatically generatedThe Save\_History method is responsible for extracting variable assignments from the self.Final\_List and saving them in the self.History list:

* Create an empty self.History list to store variable assignments.
* Initialize a boolean flag Flag\_Save as False to indicate whether to save the assignment or not.
* Iterate over each line in self.Final\_List.
* Iterate over each token in the current line.
* Check if the token represents the "Become" keyword (self.Final\_List[Line][Index][1] == "Become").
* If the "Become" keyword is found, check the tokens before and after it to determine if it represents a valid variable assignment:
* Check if the token before the "Become" keyword is a variable (self.Final\_List[Line][Index - 1][1] == "Variable").
* Check if the token after the "Become" keyword is a variable, integer, boolean, string, float, open bracket, or a negative sign (self.Final\_List[Line][Index + 1][1] in ["Variable", "Integer", "Boolean", "String", "Float","Open Bracket"] or self.Final\_List[Line][Index + 1][0] == "-").
* If the assignment is valid, set Flag\_Save to True and append a new entry to self.History containing the variable name and an empty list to store its history.
* If Flag\_Save is True, iterate over the tokens in the current line and add each token (except semicolons and commas) to the history list of the last variable assignment in self.History.
* If a semicolon or comma is encountered, set Flag\_Save back to False.
* Print the contents of self.History to display the extracted variable assignment history.

The Save\_History method extracts variable assignments from the code and saves them in self.History, preserving the order of assignment statements.

**The History Calculated:**

A screen shot of a computer program

Description automatically generated with low confidence

The calculate\_history method processes the variable assignment history stored in self.History and calculates the results of the expressions assigned to each variable:

* Create empty lists self.Results and self.Variables to store the calculated results and variable names, respectively.
* Iterate over each item in self.History, where each item represents a variable assignment.
* Extract the variable name from the item (variable = item[0]).
* Iterate over the values in the history list of the item (for i in range(len(item[1]))).
* Initialize boolean flags Flag\_Var and Flag\_Val as False.
* Iterate in reverse order over the self.Variables list to find the most recent value assigned to the current value in the history list:
* If the current value in the history list matches a value in self.Variables, set Flag\_Var to True.
* If a previous value for the current variable has a calculated result (self.Results[k][1] != None), set Flag\_Val to True.
* Replace the current value in the history list with the calculated result.
* Break the loop.
* Check the conditions to determine if the variable can be calculated:
* If Flag\_Var is True and Flag\_Val is False, break the loop because a previous value for the variable is missing.
* If either Flag\_Val is True and Flag\_Var is True or Flag\_Var is False, the expression can be evaluated.
* Join the values in the history list of the item into a string expression.
* Use eval to evaluate the expression and calculate the result.
* Append the variable and its result ([variable, result]) to self.Results.
* Append the variable to self.Variables.
* If an exception occurs during the evaluation of the expression, append ['Error', 'Input is not valid'] to self.Results and 'Variable Not Found' to self.Variables.
* Print the calculated results (self.Results).
* Call self.Memory\_Output to output the results.

The calculate\_history method calculates the results of the expressions assigned to each variable in the self.History list and stores them in self.Results.

**The Results:**

A screen shot of a computer code

Description automatically generated with low confidence

The Show\_Results method prints various results and error checks related to the code analysis process.

By printing these results and error checks, the Show\_Results method provides insights into the different stages of the code analysis process, including scanning, syntax checking, and error detection.

**The Check Output in GUI:**

A computer code on a black background

Description automatically generated with low confidence

The Check\_Output method is designed to display the results of a specific check or analysis in a text-based output area. Here's an explanation of how it works:

* self.Output\_Area.configure(state="normal"):
* This line sets the state of the output area widget to "normal" so that it can be modified.
* self.Output\_Area.delete(1.0, tk.END):
* This line deletes the existing contents of the output area, clearing it before displaying new results.
* self.Output\_Area.insert(tk.END, "Check:\n\n"):
* This line inserts the heading "Check:" followed by two new lines into the output area. It serves as a header for the displayed results.
* self.Output\_Area.insert(tk.END, '\n'.join([f"{column[0]} -> {column[1]}" for row in List for column in row])):
* This line generates a string representation of the results in the List parameter and inserts it into the output area. It iterates over each row and column in the List, and formats the content as "{column[0]} -> {column[1]}", where {column[0]} represents the first element in the column and {column[1]} represents the second element in the column. The resulting strings are joined with newline characters ('\n').
* self.Output\_Area.configure(state="disabled"):
* This line sets the state of the output area widget to "disabled" to prevent further modifications. It ensures that the displayed results cannot be edited by the user.

Overall, the Check\_Output method updates the text-based output area with the provided results, presenting them in a readable format for the user to review.

**The Memory Output in GUI:**

A picture containing text, screenshot, font

Description automatically generated

The Memory\_Output method is responsible for displaying the memory or variable assignment results in a separate text-based output area:

* self.Output\_Area2.configure(state="normal"):
* This line sets the state of the second output area widget to "normal" so that it can be modified.
* self.Output\_Area2.delete(1.0, tk.END):
* This line clears the existing contents of the second output area, preparing it for displaying new results.
* self.Output\_Area2.insert(tk.END, "Memory:\n\n"):
* This line inserts the heading "Memory:" followed by two new lines into the second output area. It serves as a header for the displayed memory results.
* self.Output\_Area2.insert(tk.END, '\n'.join([f"{row}" for row in List])):
* This line generates a string representation of the memory results in the List parameter and inserts it into the second output area. It iterates over each row in the List and formats it as a string using the f"{row}" expression. The resulting strings are joined with newline characters ('\n').
* self.Output\_Area2.configure(state="disabled"):
* This line sets the state of the second output area widget to "disabled" to prevent further modifications. It ensures that the displayed memory results cannot be edited by the user.

In summary, the Memory\_Output method updates the second text-based output area with the provided memory results, presenting them in a readable format for the user to review.