A close up of a logo

Description automatically generated

SCHOOL OF COMPUTER SCIENCES, UNIVERSITI SAINS MALAYSIA

11800 USM, PENANG, MALAYSIA

SEMESTER 1 2023/2024

CPT316 PROGRAMMING LANGUAGE IMPLEMENTATION

AND PARADIGMS

ASSIGNMENT 1 REPORT

|  |  |
| --- | --- |
| NAME | MATRIC NO. |
| MOHAMAD NAZMI BIN HASHIM | 158616 |
| MIOR MUHAMMAD IRFAN BIN MIOR LATFEE | 158450 |
| OOI JING MIN | 158768 |
| MUNIRAH BINTI SHAMSUDIN | 157518 |
| SHONEERA SIMIN | 159160 |

LECTURER: DR NIBRAS ABDULLAH AHMED FAQERA

SUBMISSION DATE: 19 NOVEMBER 2023

**Table of Contents**

[Introduction 1](#_Toc151302433)

[Language Design and Grammar 3](#_Toc151302434)

[Lexical Analysis 5](#_Toc151302435)

[Syntax Analysis 6](#_Toc151302436)

[Testing and Validation 8](#_Toc151302437)

[Conclusion 12](#_Toc151302438)

[References 13](#_Toc151302439)

[Appendix 14](#_Toc151302440)

# Introduction

Python is selected as the programming language for this assignment due to its simplicity, readability, and expressiveness. It is a perfect choice for illustrating every lexical and syntactic analysis aspect. Python's variety makes it possible to demonstrate complex language constructs clearly and effectively, which makes it possible to thoroughly explore compiler design within a well-known and widely used language.

In compiler construction, lexical analysis is the first phase of the compiler, also known as a scanner. Compilation is a crucial step in converting human-readable code into executable instructions. It converts the high-level input program into a sequence of tokens. Lexical and syntactic analysis are essential components of the main processes that a compiler does. The deterministic finite automata can be used to implement lexical analysis. A series of tokens is the output supplied to the parser for syntactic analysis. For example, the lexical analyser may identify "if" and "else" as keywords, "123" as a numeric literal or "y" as an identifier in a simple programming language. By giving the code a structured representation that makes analysis and interpretation simple, this technique lays the groundwork for the subsequent phases.

After lexical analysis, syntax analysis is usually the second step in the compilation process. Syntax analysis, sometimes called parsing, is the compiler's process of determining if the source code complies with the grammatical requirements of the target computer language. Its main goal is to ensure the token order follows the syntactic guidelines the language grammar gives. The token stream is parsed in this stage and the Abstract Syntax Tree (AST), a hierarchical representation is built. The AST illustrates the relationships between various pieces and how they are arranged, embodying the syntactic structure of the code. Syntax analysis uses a variety of parsing strategies to decipher the intricate structural details of the code, including recursive descent or parser generators like Bison or ANTLR.

Task Distribution:

|  |  |
| --- | --- |
| NAME | TASK DISTRIBUTIONS |
| MOHAMAD NAZMI BIN HASHIM | Programming: Syntax Analysis  Documentation: Syntax Analysis, testing and validation |
| MIOR MUHAMMAD IRFAN BIN MIOR LATFEE | Programming: Syntax Analysis  Documentation: Language design and grammar, testing and validation |
| OOI JING MIN | Programming: Syntax Analysis  Documentation: README file, testing and validation |
| MUNIRAH BINTI SHAMSUDIN | Programming: Lexical Analysis  Documentation: Introduction, testing and validation |
| SHONEERA SIMIN | Programming: Lexical Analysis  Documentation: Lexical Analysis, testing and validation |

# Language Design and Grammar

In this assignment, the programming language that is selected which is Python as this programming language is capable to demonstrate the compiler processes involved due to its simplicity and readability. It can be said that Python make the code more readable because it removes the uses of curly braces or keywords for the code blocks and use the indentation method. In comparison with to other code, other code usually implements the curly braces in order to create a code block or to define a function and that makes the code longer. These reasons made Python a good choice for beginners as well as professional programmers to use for the demonstration of the compiler phase processes.

Generally, Python makes it easier to show the processes of the compiler as the use of lexical analysis is made easier by Python’s concise and clear syntax. In terms of syntax analysis, Python’s syntax simplifies the implementation of the syntax analysis in order to build an Abstract Syntax Tree (AST) known as a parse tree. Due to the flexibility of Python, the grammatical rules and the structures of the language would be represented in intuitive ways, which constructing the lexical analysis and syntax analysis in an effective manner. Python also comes with an extensive standard library that includes the packages and modules that the user can use for their particular purposes. Basically, the ‘re module’ is used in this assignment, which this module is used for regular expression and for manipulating strings. The re module is important in Python in order to implement the lexical analysis due to its features that involve pattern matching and processing of strings. Hence, the packages and modules that are supported by Python help the programmers to use these for a wide range of applications including the implementation of lexical analysis and syntax analysis.

With the help of the Python module (re module), the code becomes more readable because this module provides a consistent and organized approach to the pattern matching that would be used in lexical analysis. This module is also flexible in lexical analysis because it provides a concise way to define any tokenization tasks including complex patterns. As a result, re module is very useful in terms of dealing with various sets of token values that are defined within the lexical analysis. For example, re.compile functionality is used to create a compiled regular expression within our code which are alphabet, digit, operator and punctuator.

*Figure 1 : The grammar that is defined by syntax analysis inside syntax.py (parser)*

The grammar rules that are provided above which inside the syntax analysis are essential in defining the syntax of programming language in an organize manners. Basically, syntax analysis included with the grammar rules to ensure that the language would be written in a valid way and the grammar will define the legal combinations when writing the language. For example, our main.py will allow for the input to be taken into the code and it will be processed from lexical analysis to syntax analysis. Hence, the input must be followed with the grammar rules that we already defined within the code, syntax.py. If the input is violating the grammar rules, the program will display an error and it would not go to produce the Abstract Syntax Tree because the syntax is not correct.

# Lexical Analysis

The first stage of lexical analysis is to break down the source code into understandable units called tokens. They include keywords, identifiers, operators, punctuators, and other elements necessary to comprehend the organization of the code. Lexical analysis aims to process the raw code, eliminate unnecessary components such as whitespace and comments, and locate and classify tokens for the following parsing and interpretation stages.

The process of tokenization is thorough and involves multiple steps. The lexer, first goes character by character through the code. Every figure that is encountered is bound by specific guidelines and conventions. Typically expressed in terms of finite automata or regular expressions, these rules control how tokens are identified and categorized. For example, patterns that specify their syntax are used to identify keywords (like "if," "else," and "return") or identifiers (such as variables or function names). Characters are sorted into tokens, with each ticket having a type (operator, keyword, or identifier) and associated value.

Finite automata and regular expressions are essential components of the tokenization process. These tools offer a potent way to define and spot patterns in the code. A regular expression could, for instance provide the permitted characters and their order to determine the structure of an integer or a variable name. To help the lexer detect and classify tokens according to the predefined rules, finite automata aid in recognizing these patterns.

Tokens of different types can be created, each with a unique type (token\_type) and value (value) using the Token class as a template. Meanwhile, the tokenization process is managed by the Lexer class, which has methods like advance() to traverse the code, skip\_whitespace() to disregard whitespace characters, integer() to extract numbers and identifier() to extract identifiers from the input code.

Characters are categorized into their appropriate token types according to the established rules by the categorize\_token() function, which is the central component of the lexer. It does this by appending recognized tokens into distinct categories (such as operators, punctuators, letters, integers, and keywords) that make it easier to describe the structure and elements of the input code in an ordered manner.

# Syntax Analysis

Syntax analysis is also known as parsing. It is used to analyse the token stream created during lexical analysis. The parser uses the token generated by the lexer and based it on the grammar rule created to produce the right interpretation of the source code. The grammar defined for this program are following the simple rules of arithmetic. Multiplication and division take precedence over addition and subtraction. These rules are then translated into the Abstract Syntax Tree (AST) to produce the right mathematical rules. Another grammar rule defined revolves around keywords such as “return” and “print” keywords. For instance, the “print” keyword needs to be followed by a parenthesis. Otherwise, it will return an error.

To create an abstract syntax tree, there are two ways to accomplish it, either by using a parser generator such as Yacc, Bison or ANTLR, or recursive descent parsing. This program implements the latter. Recursive descent parsing is a simpler way to build a parse as it doesn’t require any configuration. Despite its simplicity, it is very fast, robust and can support error handling. Recursive descent uses the top-down approach as it starts from the outermost grammar rule and all the way to the subexpressions and the leaves.

The process of parsing using this method will look at each of the tokens and proceed to the next token index. Most of the time, it will be looking ahead at three token indexes at once. The middle index (the second index) will be the parent node and the other two will be the left and right children respectively. The three index will be grouped together into a node in the program. It will recursively go to the “parse\_high\_precedence\_term\_prime” and “parse\_factor” functions to do the grouping of the tokens.

The result will now be returned to the main function and the program will call the “print\_ast” function to generate the AST. This function will recursively access the node created along with the children. The node accessed will be from the right of the input expression. For example, 1+2+3, it will unravel in this block (1+2) + 3. Then it will go through the block for 1+2. This will be further demonstrated in the test cases.

Some ruling created is the assignment of a variable (“parse\_statement”), mismatching of the punctuator, the skipping of parenthesis and whitespace (“parse\_expression\_prime”), and the need to have a certain token followed after a keyword (“parse\_factor”).

A few key function definitions include:

**parse\_statement:** Check when the input code is trying to assign a value to variables. After an IDENTIFIER is detected, the next token input will need to be an equal to ensure the variable assignment is valid. In addition, it checks the next token after the “print” keyword. The next token needs to be a parenthesis. For example: print().

**parse\_expression\_prime:** Check the PUNCTUATOR such as bracket and braces to ensure there is an opening and closing of the PUNCTUATOR. This function will also return the final recursive node of the tree after segregating each token into parent and children.

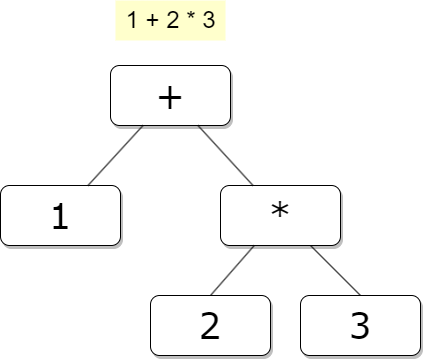
**parse\_high\_precedence\_term\_prime:** Check the precedence of arithmetic OPERATORs, such as division and multiplication.

**parse\_factor:** This function will recursively check each token along with parse\_high\_precedence\_term\_prime function and put it into a node. The node contains the parent and their children.

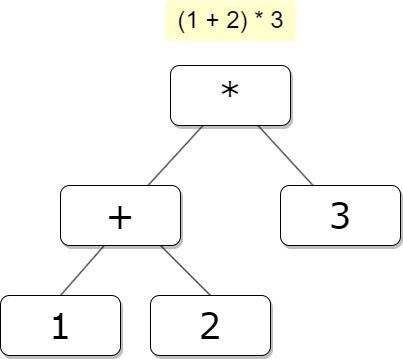
**print\_ast:** Print the abstract syntax tree. It will get the block of parents and children and disassemble the node.

# Testing and Validation

A screenshot of a computer program

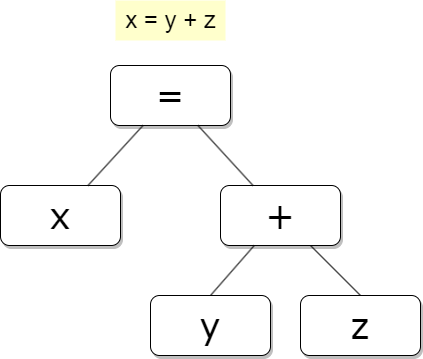
Description automatically generatedTest Case 1

The output above shows that the program prioritizes the operator that have high precedence to be executed first and this process is done by parse\_high\_precedence\_term\_prime function. Hence, 2\*3 is done first and then will be add with integer 1. This process can be seen through the AST structure graph, where the right node will be processed first before go to addition.

****A screenshot of a computer program

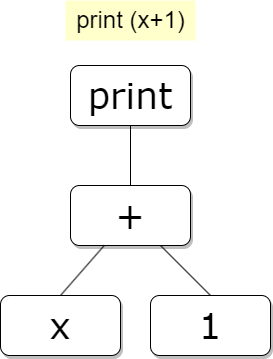
Description automatically generatedTest Case 2

The output above shows the program prioritizes the expression within the parenthesis first even though multiplication has a higher precedence than addition. The function, parse\_expression\_prime handles the prioritization of any operators that occur within the parenthesis as the parse\_expression is called recursively when an opening parenthesis is encountered. In AST, the left node will be processed first and then processed by other operators.

A screen shot of a computer

Description automatically generatedTest case 3

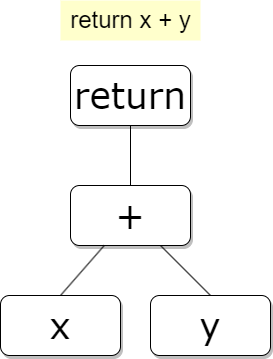
The output above shows the variable assignment with a correct syntax that already defined inside the syntax analysis. There is selection method within the parse\_statement function, in which it will recognize the beginning of the variable assignment if the current token type is identifier. This function will check the variable name and consume it, then it will check for assignment operator (=). If there is an assignment operator, the program will parse the expression at the right of the assignment operator by using parse\_expression function.

Test Case 4A screen shot of a computer program

Description automatically generated

The output above shows the print statement with the right syntax defined within the parser of the program. Within the parse\_statement function, the function checks the ‘print’ keywords, if the ‘print’ is encountered, it expects the opening parenthesis following by an expression with closing parenthesis. If both of the parenthesis present, the program will continue to produce the AST graph and the parenthesis will be included in the AST because the parenthesis is being consumed within the function.

A screen shot of a computer

Description automatically generatedTest Case 5

The output above shows the return statement with the correct syntax following the grammar defined within the syntax analysis. Within the parse\_statement function, there is a selection method, where the parser will check whether the current token value is ‘return’ and it will recognize it as a return statement. It will consume the return, which means the index will advance to the next index and call parse\_expression function to handle the expression of x+y.

A screen shot of a computer

Description automatically generatedTest Case 6

The output above shows an error message is displayed when the input above is parse through the program. It is because the syntax for print statement has been defined which is the print statement must be follow with the parentheses (opening and closing). If one of these parentheses is absent, then the program will display the messages either missing opening or closing parentheses. This process is done by the parse\_statement function. Hence, the correct syntax is print(x+1).

A screenshot of a computer program

Description automatically generatedTest Case 7

The output above shows that error message will be displayed when the input contains ‘OTHER’ tokens. Within the parse\_factor function, it will check the condition related with the tokens that are defined in lexical analysis and the error messages will be displayed indicating that the unexpected token is encountered.

**A screenshot of a computer program

Description automatically generated**Test Case 8

The output above shows that an error message is displayed when the input does not follow the grammar defined in the syntax analysis. This error is displayed because there are no grammar rules that are defined to handle the ‘++’ operator. Hence, the correct syntax would be 1+2 instead of 1++2.

# Conclusion

In conclusion, the development of this program has provided significant understanding of the process of the lexical and syntax analysis for a simple programming language. This was successfully done by generating tokens and producing the right syntax output of a source code. Our program provides coverage on the mathematical expressions and assignment of variables. Through this, we learnt to categorize the right lexeme and separation of different types of tokens. The tokens generated from the lexical analysis are then used to give us the right order of the Abstract Syntax Tree (AST) by going into the parser and grammatical rulings.

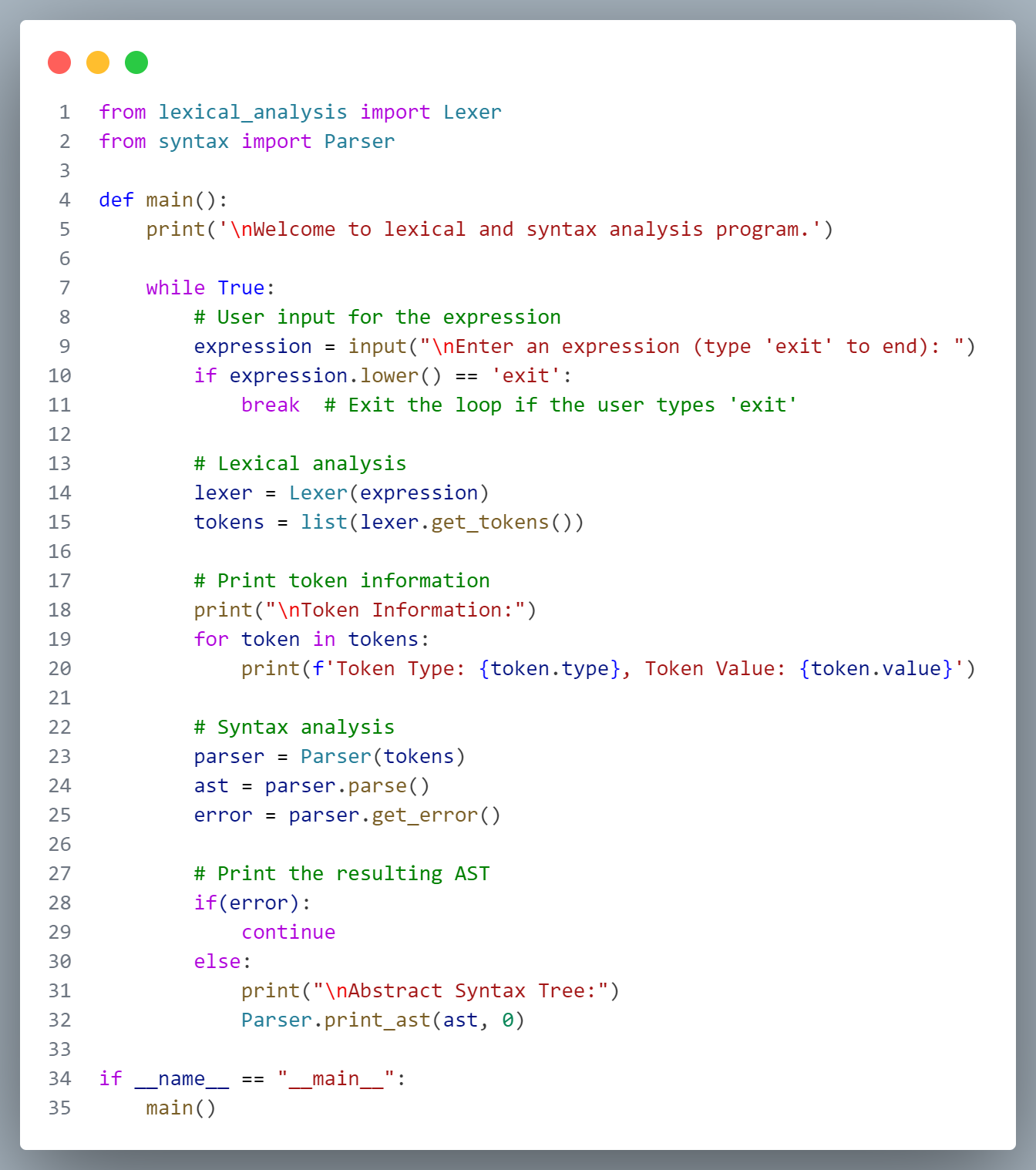
We have encountered some problems when dealing with more complex operations and refining the error-handling mechanisms. Fortunately, there are a few references that help us to solve these problems. As we reflect on the process of designing and implementing the program, we found multiple aspects for further improvement of the rules created. Thus, we did the refinement bit by bit over the duration of the assignment. Finally, we hope to further explore the areas of compiler analysis in the future.

# References

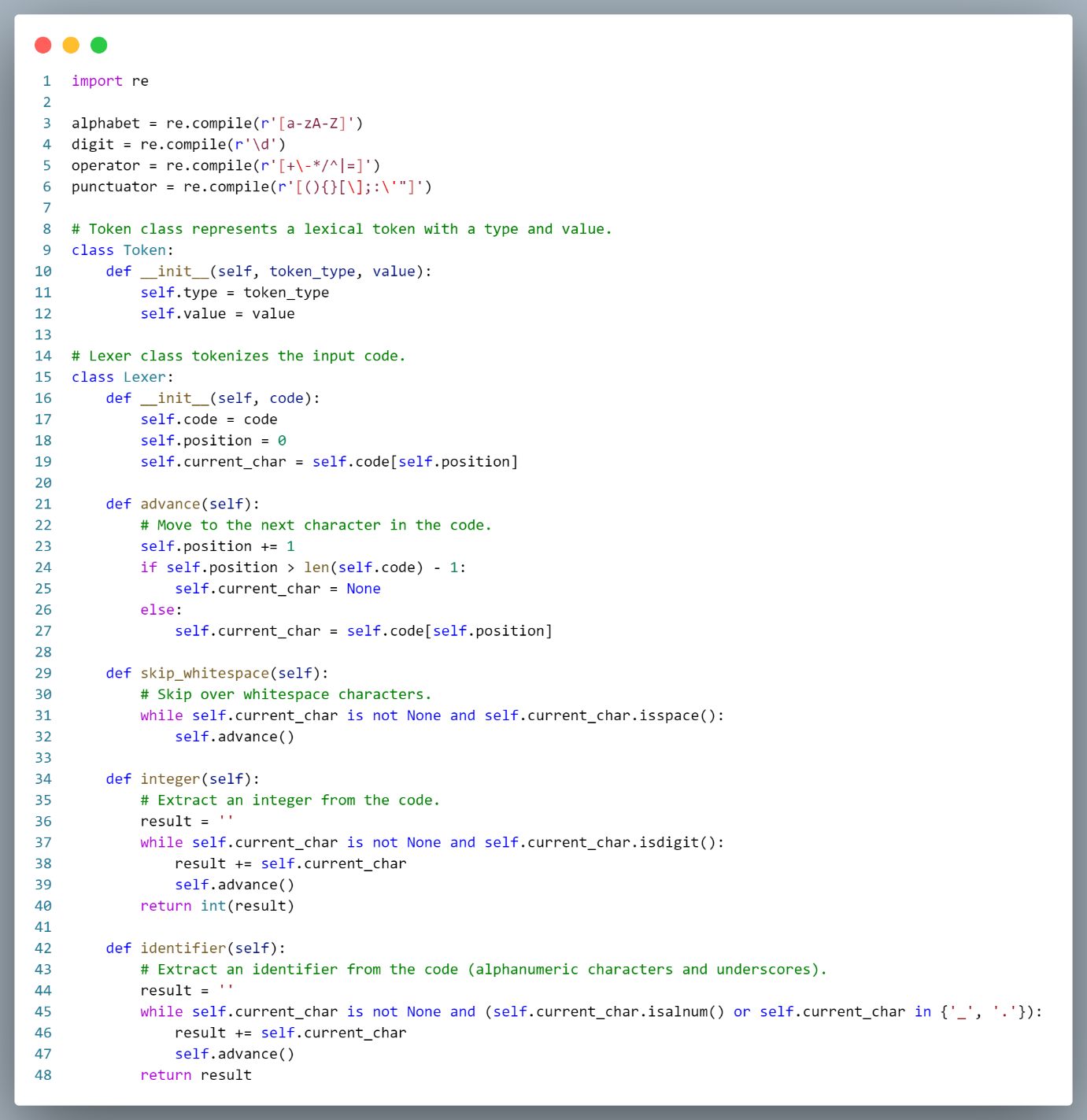
1. Nystrom, R. (2021). Crafting interpreters. Genever Benning.
2. Lexical analysis and syntax analysis. (2023, January 24). GeeksforGeeks. <https://www.geeksforgeeks.org/lexical-analysis-and-syntax-analysis/>
3. GeeksforGeeks. (2023, September 27). *Introduction of lexical analysis*. <https://www.geeksforgeeks.org/introduction-of-lexical-analysis/>
4. Kucuk, Z. (2023, September 22). *Overview of the python programming language*. Medium. https://zynpkucuk.medium.com/overview-of-the-python-programming-language-fb76a1dab200

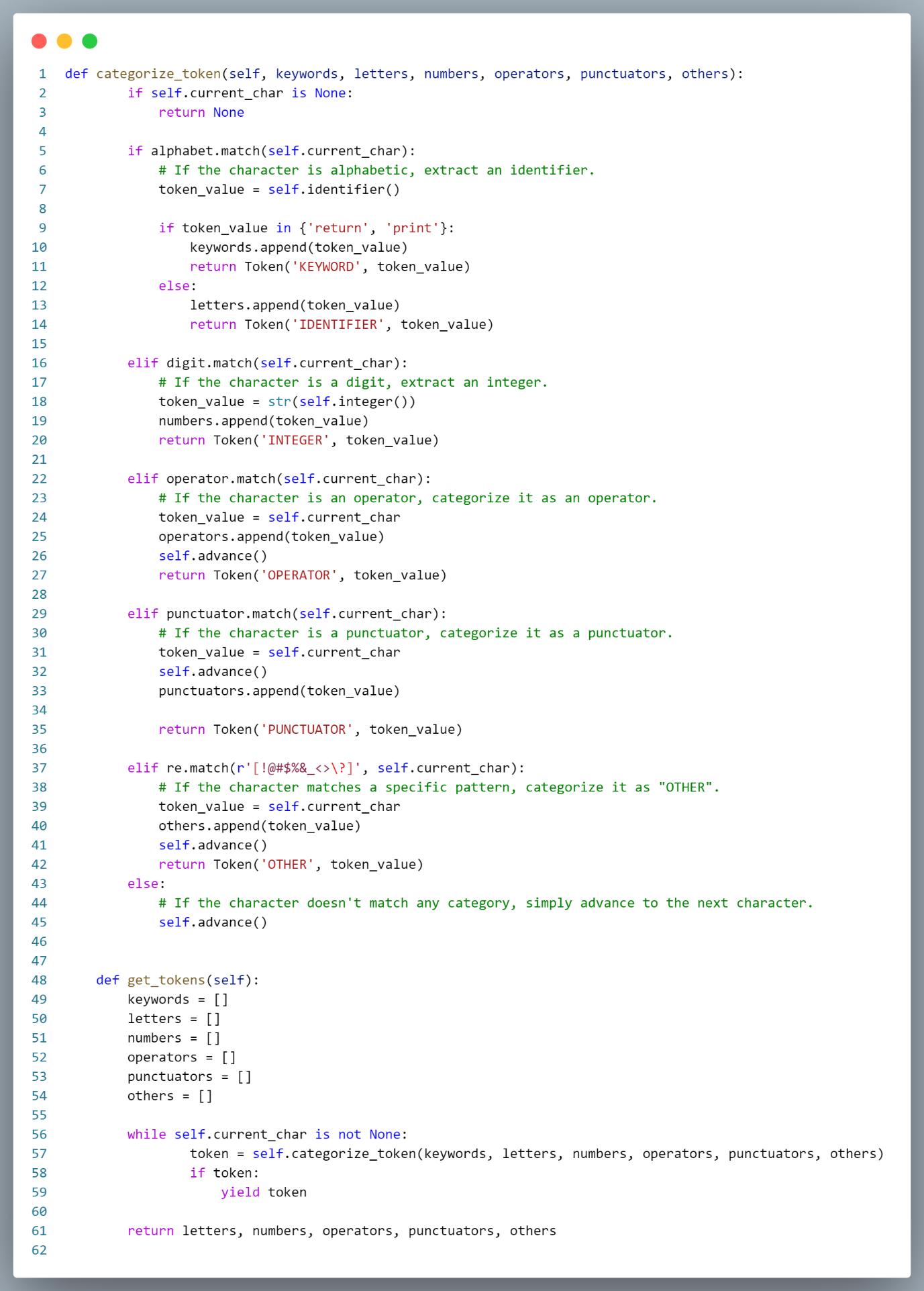
# Appendix

**main.py**



**Lexical\_analysis.py**





**Syntax.py**

