1.0 Introduction

1.1 Introduction to compiler

A compiler is acknowledged in the broad field of programming as the crucial component that connects machine-executable instructions to source code that can be read by humans. A compiler is a language translator that moves through several stages, all of which help to convert abstract ideas into a format that can be processed by a computer's processor [6]. The preliminary steps, particularly the lexical and syntactic analysis, are critical to this complex procedure.

Lexical analysis is the process of carefully examining the source code and breaking it down into a series of tokens. [1] Tokens are the least important parts of the programming language; they include literals, operators, keywords, and identifiers. The main goal is to transform understandable code into a format that can be handled more efficiently by later compiler stages [6]. In this stage, regular expressions and finite automata are tools that define patterns that characterise the syntactic structure of different tokens.

The next step, syntax analysis, consists of parsing the token stream that is produced by lexical analysis. Building a parse tree or, in this case, an abstract syntax tree (AST) is the main goal [7]. This tree illustrates the source code's hierarchical structure and guarantees that it complies with the programming language's grammar rules. The AST helps identify language constructs and serves as a roadmap for further compiler operations.

1.2 Introduction to BASIC language

Beginner's All-purpose Symbolic Instruction Code, or BASIC, is an abbreviation for a programming language that is well-known for being easy to learn and use. Originally created with the novice user in mind, BASIC is a is a programming language well-known for being accessible and user-friendly. Originally designed to be user-friendly, BASIC has developed into a flexible language with a wide range of applications. Because of its emphasis on simplicity in design, it's a great place for beginners to start learning programming [6].

John G. Kemeny and Thomas E. Kurtz were the pioneers of BASIC in the mid-1960s. Their aim was to democratize access to computing power by creating a language that could overcome the technological barriers that were prevalent in the computer environment at that time. [6]. The advent of BASIC was groundbreaking because it made direct computer interaction possible for people who had no prior experience with programming.

Not only is BASIC significant historically, but it also leaves behind an enduring record of simplicity and learnability. BASIC has a syntax that minimises obstacles for novices and emphasises readability. The commands in BASIC are simple and resemble those in English.

```
10 PRINT "HELLO, WORLD!"
.
.
.
.
20 END
```

Based on the code snippet above, line numbers act as a program's structural guide in this succinct example. The programme ends when the *END* command is used, and the *PRINT* statement prints text on the screen. Due to the simplicity of these commands, BASIC is an excellent starting point for people who are new to programming concepts. [6].

Our study provides a unique opportunity to explore the complexities of Python compiler design, focused on a dialect of BASIC. Converting high-level source code into machine-readable instructions is known as compiler construction, and our attention to a particular BASIC version highlights the usefulness of lexical and syntax analysis in the Python programming language [10]. By utilising Python's flexibility, we can handle the challenges of compiler design and demonstrate how this language can be used to build a compiler for a language that is domain-specific, such as BASIC. This demonstrates not only our dedication to real-world implementation but also Python's versatility in a variety of programming scenarios.

2.0 Lexical Analysis

One of the most important steps in creating a compiler for the BASIC programming language is the lexical analysis phase. It acts as the first stage in decomposing the source code into a series of tokens through analysis. Tokens are the smallest meaningful parts of a programming language, such as operators, literals, keywords, and identifiers. The main purpose of the lexical analysis phase is to transform the human-readable source code into a format that can be handled more effectively by the compiler in the later stages. The input source code is scanned in this procedure, character by character and then the characters are grouped into tokens according to preset patterns and criteria.

2.1 Regular Expressions

Regular expressions serve an important role in lexical analysis which it is used to create patterns in programming languages that depict the syntactic structure of various tokens.

In our compiler,

The regular expressions **defined** consists of the following:

Number: [1-9]+[0-9]|0*

This regular expression is used to specify for **positive integer including zero**.

Keyword: import|from|if|elif|else|return|def|break|continue

This regular expression is used to specify for **keywords** including "import," "from," "if," "elif," "else," "return," "def," "break," and "continue".

*Identifier: [a-zA-Z_]+[a-zA-Z_0-9]**

This regular expression is used to specify for **names of variables and functions**. It accepts string that begins with alphabet and ends with alphabet or integer.

String literal: ".?"*

This regular expression is used to specify for **string that contain any character** of finite length.

Operator: [+\-/%]*

This regular expression is used to specify for **arithmetic operators** '+', '-', '*', '/', and '%'.

Equals: =

This regular expression is used to specify for the **equal** symbol.

Open parenthesis: (

This regular expression is used to specify for **open parenthesis** '('.

Close parenthesis:)

This regular expression is used to specify for **close parenthesis** ')'.

Space: [\t\r]

This regular expression is used to specify for **carriage returns and tabs**.

2.2 Finite Automata

Finite automata are important for specifying regular expressions in the lexical analysis stage of a compiler because they identify and handle patterns in the source code. These automata find patterns in input strings by establishing states that correspond to different criteria given by regular expressions. The finite automata speed up the process of tokenizing the source code by quickly recognizing and classifying substrings that match certain regular expression patterns [12].

The finite automata created for our compiler involves inputs, states, and transitions to specify for each regular expression defined. There is a total of 23 states included in the finite automata which the first state is for receiving input from the user, the second and third states are used for identifying numbers, 4 to 12 states are used for identifying keywords, 13 state is used for identifying identifiers, 14 to 18 states are used for identifying operators, 19, 20 and 21 are each for identifying equal, open parentheses and close parentheses symbol and lastly 22 to 23 states are used for identifying space.

The construction of Finite State Automata for a BASIC compiler only utilizes Deterministic Finite Automata based on the given regular expressions. As a result, the procedure entails creating DFAs straight from the regular expressions, ensuring predictable behaviour in token identification during the BASIC compiler's lexical analysis stage.

The following diagrams show finite automata created based on each regular expression defined in Section 2.1:

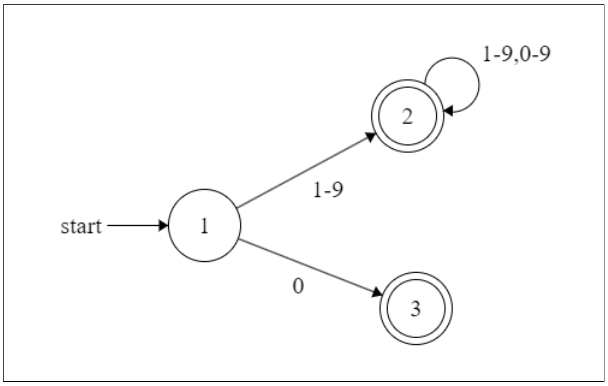


Figure 2.2.1 This figure depicts the automata which will accept positive integer including zero

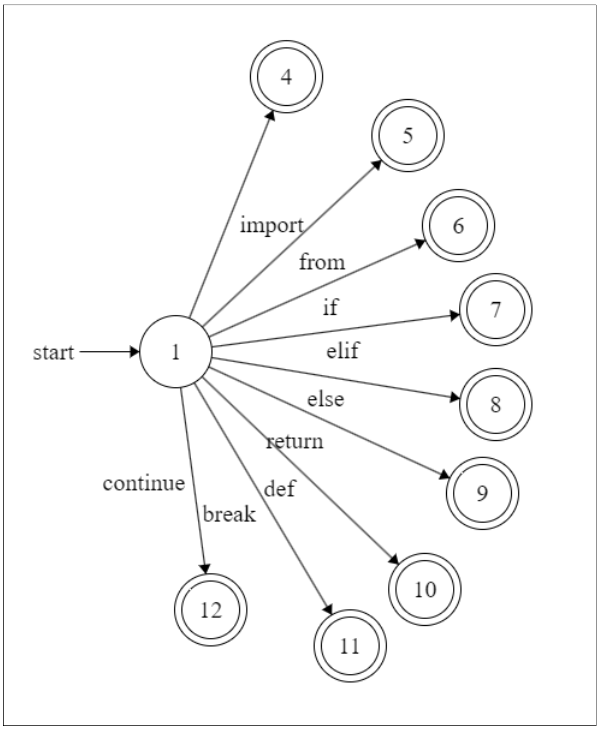


Figure 2.2.2 This figure depicts the automata which will accept keywords including "import," "from," "if," "elif," "else," "return," "def," "break," and "continue"

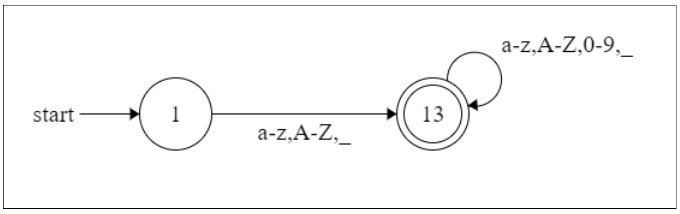


Figure 2.2.3 This figure depicts the automata which will accept identifiers that are string that begins with alphabet or underscore and ends with alphabet, integer or underscore

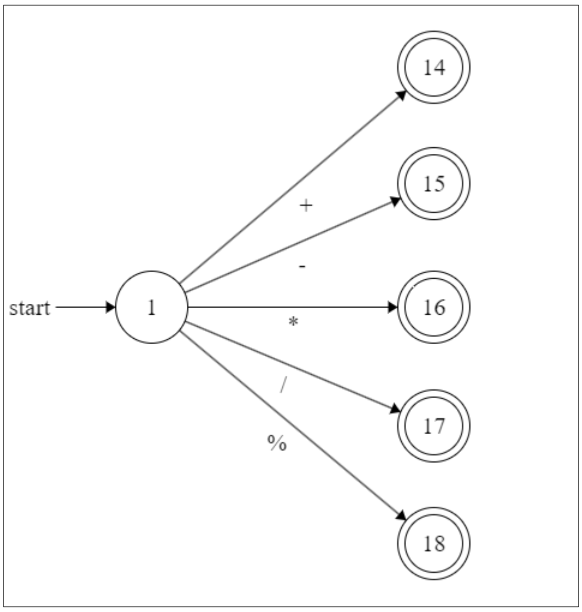


Figure 2.2.4 This figure depicts the automata which will accept arithmetic operators '+', '-', '', '/', and '%'.*

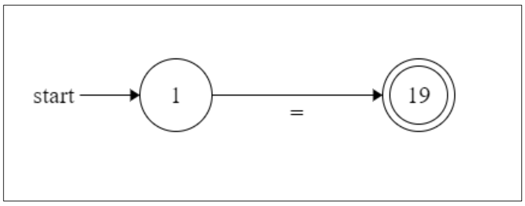


Figure 2.2.5 This figure depicts the automata which will accept equal symbol

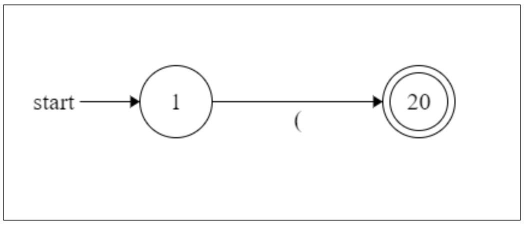


Figure 2.2.6 This figure depicts the automata which will accept open parenthesis

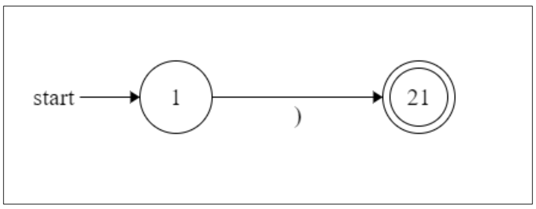


Figure 2.2.7 This figure depicts the automata which will accept close parenthesis

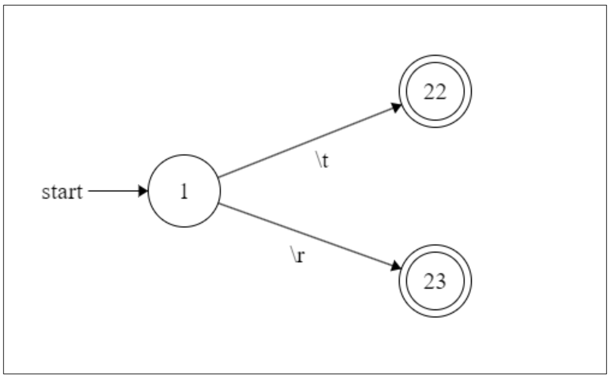


Figure 2.2.8 This figure depicts the automata which will accept carriage returns and tabs

2.3 Code Reference

There are two classes involved in the lexical analysis phase which are the *TokenType* and the *Token* class.

```
# Different token types for Enum
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    STRING_LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    SPACE = auto()
    EOF = auto()
```

Figure 2.3.1 The figure shows the code snippet for the class TokenType

```
# Class 'Token' represent a token with their type and value
class Token:
    def __init__(self, type: TokenType, value: str) -> None:
        self.type: TokenType = type
        self.value: str = value

    def __repr__(self):
        return f'{{ {self.type}, {self.value} }}'
```

Figure 2.3.2 The figure shows the code snippet for the class Token

The *Token* class **create tokens based on the input program file**. It comes with a constructor that accepts two parameters including a type for the *TokenType* class and a value for respective token. The token type and token value both have an attribute in the class. The *__repr__* method in the *Token* class function for **producing a readable string representation of the token** by enclosing its type and value in curly braces. The tokenize function plays a main role in the tokenization process which iterates over the code string using a while loop and looks for patterns defined in a *TOKEN_REGEX* class. A new instance of the token is produced and added to the tokens list for every successful match. An exception will be raised if the input character is invalid showing that there isn't a match for the current character in the code string. An end-of-file *EOF* token will be added to the token list for informing the parser that there are no more tokens to process. After processing the complete code string, the method produces and returns the list of tokens.

The *TokenType* class uses enumeration for **specifying type for each token**. Every member of the enumeration is automatically assigned a unique value using the *auto()* function from the *Enum* class. The *TOKEN_REGEX* is being used to specify key that denotes a different type of token and the accompanying value is a regular expression pattern that specifies the token's syntax.

At the end of the *lexer.py* file, there is a *while* loop serves as the entry point for the compiler. It prompted user to input the program file or code to be compiled. User input will then be passed to the tokenize function and the list of tokens that are produced will be stored in the tokens variable. Following is a *for* loop that has been used to iterates through the token list to output respective token by calling the __repr__ method of the *Token* class.

3.0 Syntax Analysis

Constructing a parse tree is a necessary step in the syntax analysis process to identify the input program's syntax and make sure it complies with the language's grammar [2]. In the lexical analysis phase, we can recognize the token for each element of the input and store them in symbol table:

- The **arithmetic operators**: *PLUS ('+'), MINUS ('-'), MUL ('*'), DIV ('/') and MOD (%)*
- The **positive integer including zero**: *0...9*
- The **keywords** including *"import," "from," "if," "elif," "else," "return," "def," "break," and "continue"*
- The **identifiers** that are *string that begins with alphabet and ends with alphabet or integer*
- The **equal symbol***: ' = '*
- The **open parenthesis:** *' ('*
- The **close parenthesis:** *') '*
- The **carriage returns and tabs:** *' \r ' and ' \t '*

3.1 Parsing

In our parser,

It can **recognize basic arithmetic expressions with positive integer including zero** by using the token stream generated by the lexical analysis as done in the lexical analysis phase (Section 2.0). These are the productions of **Context-Free Grammar** (CFG):

*Program --> Stmt**

Stmt: --> AdditiveExpr

*AdditiveExpr --> MultiplicateiveExpr ((PLUS | MINUS) MultiplicateiveExpr)**

*MultiplicativeExpr --> Factor ((MUL | DIV) Factor) **

Factor --> Integer

| (Open_Paren) AdditiveExpr (Close_Paren)

In this case, the first token in the program will be an integer number and may be followed by mathematical operator. To achieve this CFG in our program, we use **recursive descent parser** which is a top-down parser where each such procedure implements one of the non-terminals of the grammar. *Parser* class uses the tokens in the symbol table to generate an abstract syntax tree (AST). At first, *eat* class will read and return the token and move to the next token. If the token type is not matches the expected type where the parser is expecting to find in the input program, then it will raise an error exception.

```
def eat(self) -> Token:
  return self.tokens.pop(0)

def expect(self, tokentype: TokenType) -> Token:
  if self.tk().type == tokentype:
    return self.eat()

  raise Exception(__file__, 'Token expected ', tokentype.name, ' not found. Get ', self.tk(), ' instead.')
```

*Figure 3.1.1 The figure shows the code snippet of methods **eat and expect** and the error exception done by **Exception***

The following list of methods from the *parser* class allows it to identify input tokens and alert the user to any syntax issues:

```
def parse_program(self) -> Program:
    program = Program()
    while self.tk().type != TokenType.EOF:
        program.body.append(self.parse_stmt())
    return program

def parse_stmt(self) -> Stmt:
    return self.parse_additive_expr()

def parse_additive_expr(self) -> Stmt:
    left = self.parse_multiplicative_expr()
    while self.tk().value == '+' or self.tk().value == '-':
        operator = self.eat().value
        right = self.parse_multiplicative_expr()
        left = BinaryExpr(left, operator, right)
    return left

def parse_multiplicative_expr(self) -> Stmt:
    left = self.parse_factor()
    while self.tk().value == '*' or self.tk().value == '/':
        operator = self.eat().value
        right = self.parse_factor()
        left = BinaryExpr(left, operator, right)
    return left

def parse_factor(self) -> Stmt:
    if self.tk().type == TokenType.NUMBER:
        return NumberFactor(int(self.eat().value))
    elif self.tk().type == TokenType.OPEN_PAREN:
        self.eat()
        value = self.parse_additive_expr()
        self.expect(TokenType.CLOSE_PAREN)
        return value

    raise Exception(__file__, 'Cannot not parse current token.', self.tk())
```

*Figure 3.1.2 The figure shows the code snippet of methods **parse_program, parse_stmt, parse_additive_expr, parse_multiplicative_expr and parse_factor** and error exception done by the **Exception***

The program starts to parse the *parse_program* which defines zero or more statement (*Stmt**). It will use the *while* loop to parse the statement until it reaches the end-of-file (*EOF*). In *parse_stmt* method, it will parse the statement and then call the *parse_additive_expr*. The *parse_additive_expr* method will parse an additive expression (*AdditiveExpr*) and use the *while* loop to detect the arithmetic operator *PLUS* or *MINUS*, then it will construct a binary expression for this operator. The *parse_multiplicative_expr* method will parse a multiplicative expression (*MultiplicativeExpr*) and use the *while* loop to detect the arithmetic operator *MUL* and *DIV*, then it will construct a binary expression for this operator.

In *parse_factor* method, the parser will parse a factor which can be either an integer (*Factor -> Integer*) or an expression enclosed in parentheses (*Factor -> OPEN_PAREN expr CLOSE_PAREN*). Therefore, the parser should choose the parsing path based on the current token type. It constructs a *NumberFactor* for integer values and handles parentheses by recursively call *parse_additive_expr* method.

3.2 Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) is built as it takes our tokens and turns them into a tree that shows the real code structure [2]. It ensures that a list of token has valid syntax according to the grammar rules. The structure of each node in the AST that we build is described in *ast.py*. Every node stands for a different language element, such as a *program*, *stmt*, *factor*, *binary_expr* and *number_factor*.

The AST node type in the program:

```
class NodeType(Enum):
    PROGRAM = auto()
    STMT = auto()
    FACTOR = auto()
    BINARY_EXPR = auto()
    NUMBER_FACTOR = auto()
```

*Figure 3.2.1 The figure depicts the code snippet of class **NodeType** which contains all necessary functions to build and Abstract Syntax Tree*

The code in the *ast.py* has the function to build the ASTs to store each expression that the parser recognise, so that it can traverse it recursively to deal with the grammar language that our group defined to check the precedence of the mathematical operators

Here is an example (2 + 5 * 3).

Derivation:

Program

-> Stmt

-> AdditiveExpr

-> MultiplicativeExpr PLUS MultiplicativeExpr

-> Factor MUL Factor PLUS MultiplicativeExpr

-> Integer MUL Factor PLUS MultiplicativeExpr

-> Integer MUL Integer PLUS MultiplicativeExpr

-> Integer MUL Integer PLUS Factor

->Integer MUL Integer PLUS Integer

Parse Tree:

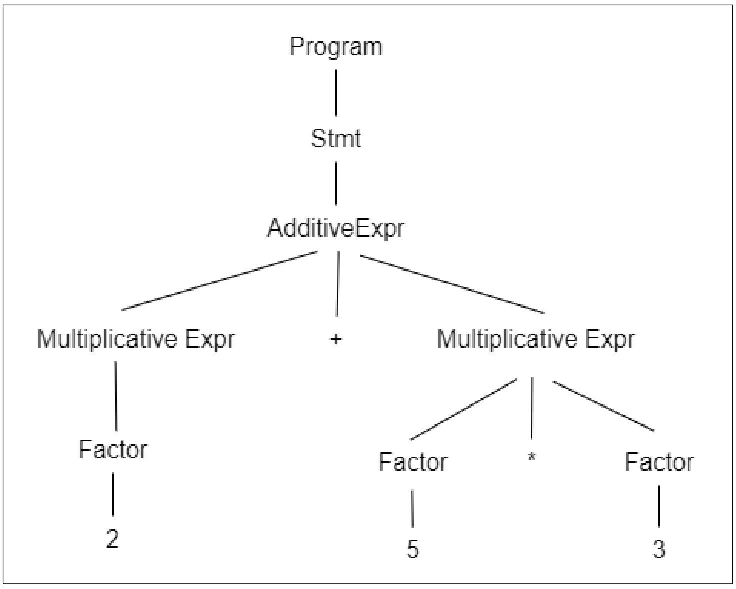


Figure 3.2.2 The figure depicts the Parse Tree generated after running the code in ast.py

Abstract Syntax Tree:

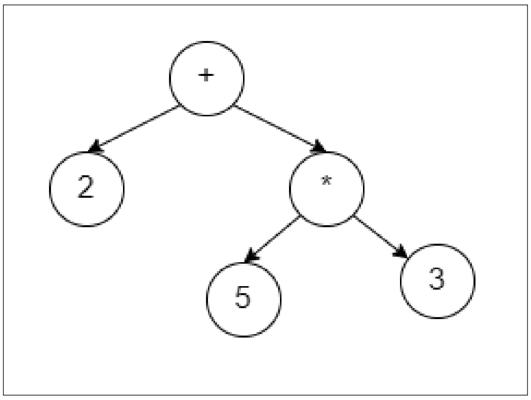


Figure 3.2.3 The figure depicts the Abstract Syntax Tree generated after running the code in ast.py

The program generates the **AST** at the source program:

```
{
        PROGRAM
        {
                BINARY_EXPR
                { NUMBER_FACTOR, 2 }
                +
                {
                        BINARY_EXPR
                        { NUMBER_FACTOR, 5 }
                        *
                        { NUMBER_FACTOR, 3 }
                }
        }
}
```

Figure 3.2.3 The figure depicts the Parse Tree generated via the compiler

```
{

        {

                { 2 }
                +
                {

                        { 5 }
                        *
                        { 3 }
                }
        }
}
```

Figure 3.2.4 The figure depicts the Abstract Syntax Tree generated via the compiler

The *inner* curly bracket represents that the program will execute it first then execute the *outer* curly bracket. In parse tree, it will perform form the *bottom* first, then pass the result up to the *root* to perform the rest. Therefore, when we traverse the tree, we would perform 5 * 3 first, then add the result to 2. As a conclusion, in syntax analysis, it parses the token stream generated by the lexical analysis and implements a parser (recursive descent) to generate an abstract syntax tree (AST). The AST constructed in our program is proves that the input program adheres to the grammar rules.

4.0 Test Cases

4.1 Lexical Analysis *(lexer.py)*

Tokenization will occur during the lexical analysis stage of the process. The input that contains undefined token types will also result in an invalid token error. The regular expression accepted in this phase are positive integer including zero, keywords, identifiers, arithmetic operators, equal symbol, open parenthesis, close parenthesis, carriage returns and tabs.

4.1.1 Test Case 1

Terminal Input: *5+7*10-3*

lexer.py

Lexical & Syntax > Lexical Analysis > lexer.py > tokenize

```
from enum import Enum, auto
import re

# Different token types for Enum
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    STRING_LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
PS C:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)>  & 'C:\Users\MyAcer\anaconda3\python.exe' 'c:\Users\MyAcer\.vscode\extensions\ms-python.python-2023.20.0\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '58603' '--' 'c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analysis\lexer.py'
Lexical Analysis:
>>> 5+7*10-3
{ TokenType.NUMBER, 5 }
{ TokenType.OPERATOR, + }
{ TokenType.NUMBER, 7 }
{ TokenType.OPERATOR, * }
{ TokenType.NUMBER, 10 }
{ TokenType.OPERATOR, - }
{ TokenType.NUMBER, 3 }
```

4.1.2 Test Case 2

Terminal Input: *Aplus = 80%(3+2)+"Excellent"*

lexer.py

Lexical & Syntax > Lexical Analysis > lexer.py > ...

```
    EQUALS = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    SPACE = auto()
    EOF = auto()

# Regular expressions for each type of token
TOKEN_REGEX = {
    TokenType.NUMBER: r'[1-9]+[0-9]*|0',
    TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
    TokenType.IDENTIFIER: r'[a-zA-Z_]+[a-zA-Z_0-9]*',
    TokenType.STRING_LITERAL: r'".*?"',
    TokenType.OPERATOR: r'[+\-*/%]',
    TokenType.EQUALS: r'=',
    TokenType.OPEN_PAREN: r'\(',
    TokenType.CLOSE_PAREN: r'\)',
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Lexical Analysis:
>>> Aplus = 80%(3+2)+"Excellent"
{ TokenType.IDENTIFIER, Aplus }
{ TokenType.EQUALS, = }
{ TokenType.NUMBER, 80 }
{ TokenType.OPERATOR, % }
{ TokenType.OPEN_PAREN, ( }
{ TokenType.NUMBER, 3 }
{ TokenType.OPERATOR, + }
{ TokenType.NUMBER, 2 }
{ TokenType.CLOSE_PAREN, ) }
{ TokenType.OPERATOR, + }
{ TokenType.STRING_LITERAL, Excellent }
```

4.1.3 Test Case 3

Terminal Input: *x = 7+10/2*(2+8)*

```
lexer.py ×
Lexical & Syntax > Lexical Analysis > lexer.py > ...
from enum import Enum, auto
import re

# Different token types for Enum
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    STRING_LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS          Python Debug Console

M 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)'; & 'C:\Users\MyAcer\anaconda3\python.exe' 'c:\Users\MyAcer\.vsco
de\extensions\ms-python.python-2023.20.0\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '62246' '--' 'c:\Users\MyAcer
\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analysis\lexer.py'
Lexical Analysis:
>>> x = 7+10/2*(2+8)
{ TokenType.IDENTIFIER, x }
{ TokenType.EQUALS, = }
{ TokenType.NUMBER, 7 }
{ TokenType.OPERATOR, + }
{ TokenType.NUMBER, 10 }
{ TokenType.OPERATOR, / }
{ TokenType.NUMBER, 2 }
{ TokenType.OPERATOR, * }
{ TokenType.OPEN_PAREN, ( }
{ TokenType.NUMBER, 2 }
{ TokenType.OPERATOR, + }
{ TokenType.NUMBER, 8 }
{ TokenType.CLOSE_PAREN, ) }
```

4.1.4 Test Case 4 – Error Exception

Terminal Input: *&*

```
lexer.py ×
Lexical & Syntax > Lexical Analysis > lexer.py > ...
from enum import Enum, auto
import re

# Different token types for Enum
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    STRING_LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    SPACE = auto()
    EOF = auto()

# Regular expressions for each type of token
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS          Python Debug Console

Lexical Analysis:
>>> &
Traceback (most recent call last):
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analys
is\lexer.py", line 78, in <module>
    tokens = tokenize(code)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analys
is\lexer.py", line 70, in tokenize
    raise Exception(__file__, 'Not valid character: ', code[0])
Exception: ('c:\\Users\\MyAcer\\Desktop\\SEM 5\\CPT316\\Assignment 1\\Final\\Lexical & Syntax (AST and Parse Tree)\\Lexical & Syntax\
\Lexical Analysis\\lexer.py', 'Not valid character: ', '&')
```

4.1.5 Test Case 5 – Error Exception

Terminal Input: *final$*

lexer.py ×

Lexical & Syntax > Lexical Analysis > lexer.py > ...

```python
    SPACE = auto()
    EOF = auto()

# Regular expressions for each type of token
TOKEN_REGEX = {
    TokenType.NUMBER: r'[1-9]+[0-9]*|0',
    TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
    TokenType.IDENTIFIER: r'[a-zA-Z_]+[a-zA-Z_0-9]*',
    TokenType.STRING_LITERAL: r'".*?"',
    TokenType.OPERATOR: r'[+\-*/%]',
    TokenType.EQUALS: r'=',
    TokenType.OPEN_PAREN: r'\(',
    TokenType.CLOSE_PAREN: r'\)',
    TokenType.SPACE: r'[ \t\r]',
}

# Class 'Token' represent a token with their type and value
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Lexical Analysis:
>>> final$
Traceback (most recent call last):
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analys
is\lexer.py", line 78, in <module>
    tokens = tokenize(code)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Lexical Analys
is\lexer.py", line 70, in tokenize
    raise Exception(__file__, 'Not valid character: ', code[0])
Exception: ('c:\\Users\\MyAcer\\Desktop\\SEM 5\\CPT316\\Assignment 1\\Final\\Lexical & Syntax (AST and Parse Tree)\\Lexical & Syntax\
\Lexical Analysis\\lexer.py', 'Not valid character: ', '$')
```

Since '$' character is not accepted, therefore the program will raise the error exception and prompt the message to remind user that '$' is not a valid character.

4.2 Syntax Analysis *(main.py)*

In the syntax analysis phase, a parse tree and Abstract Syntax Tree (AST) will be generated as a product of the tokenization process in the lexical analysis process. The language defined in this phase is **basic arithmetic expressions with positive integer including zero.**

4.2.1 Test Case 1

Terminal Input: *5+7*10-3*

main.py ×

Lexical & Syntax > Syntax Analysis > main.py > ...

```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   Python Debug Console

```
Current working directory: C:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)
Syntax Analysis:
>>> 5+7*10-3
Parse Tree:
{
        PROGRAM
        {
                BINARY_EXPR
                {
                        BINARY_EXPR
                        { NUMBER_FACTOR, 5 }
                        +
                        {
                                BINARY_EXPR
                                { NUMBER_FACTOR, 7 }
                                *
                                { NUMBER_FACTOR, 10 }
                        }
                }
                -
                { NUMBER_FACTOR, 3 }
        }
}

Abstract Syntax Tree:
{
        {
                {
                        {5}
                        +
                        {
                                {7}
                                *
                                {10}
                        }
                }
                -
                {3}
        }
}
```

4.2.2 Test Case 2

Terminal Input: *3+4-6*2*

main.py

Lexical & Syntax > Syntax Analysis > main.py > ...

```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Syntax Analysis:
>>> 3+4-6*2
Parse Tree:
{
        PROGRAM
        {
                BINARY_EXPR
                {
                        BINARY_EXPR
                        { NUMBER_FACTOR, 3 }
                        +
                        { NUMBER_FACTOR, 4 }
                }
                -
                {
                        BINARY_EXPR
                        { NUMBER_FACTOR, 6 }
                        *
                        { NUMBER_FACTOR, 2 }
                }
        }
}

Abstract Syntax Tree:
{

        {

                {

                        {3}
                        +
                        {4}
                }
                -
                {

                        {6}
                        *
                        {2}
                }
        }
}
```

4.2.3 Test Case 3

Txt File Input: *test.txt*

Step-by-Step Approach:

1. Input *file* at terminal to enter the path to the desired *.txt* file.
2. Copy the *.txt* file directory and paste it on the terminal.
3. Click *Enter* to read the *.txt* file and produce the parse tree and abstract syntax tree.

main.py × test.txt

Lexical & Syntax > Syntax Analysis > main.py > ...

```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
    parser = LexicalSyntax.Parser()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Syntax Analysis:
>>> file
Enter the path to the .txt file: C:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexica
l & Syntax\Syntax Analysis\test.txt


Parse Tree:
{
        PROGRAM
        {
                BINARY_EXPR
                { NUMBER_FACTOR, 2 }
                +
                {
                        BINARY_EXPR
                        { NUMBER_FACTOR, 5 }
                        *
                        { NUMBER_FACTOR, 3 }
                }
        }
        {
                BINARY_EXPR
                { NUMBER_FACTOR, 4 }
                *
                { NUMBER_FACTOR, 5 }
        }
        {
                BINARY_EXPR
               {
                       BINARY_EXPR
                       { NUMBER_FACTOR, 1 }
                       +
                       { NUMBER_FACTOR, 2 }
               }
               *
               { NUMBER_FACTOR, 3 }
        }
}

Abstract Syntax Tree:
{

        {

                {2}
                +
                {

                        {5}
                        *
                        {3}
                }
        }
        {

                {4}
                *
                {5}
        }
        {

                {

                        {1}
                        +
                        {2}
                }
                *
                {3}
        }
}
```

4.2.4 Test Case 4 – Error Exception

Terminal Input: *j = 3+r*

main.py

Lexical & Syntax > Syntax Analysis > main.py > ...

```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
    parser = LexicalSyntax.Parser()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Syntax Analysis:
>>> j = 3+r
Traceback (most recent call last):
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 67, in <module>
    repl()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 48, in repl
    program = parse_input(user_input)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 11, in parse_input
    program = parser.parse(input_text)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 30, in parse
    return self.parse_program()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 41, in parse_program
    program.body.append(self.parse_stmt())
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 49, in parse_stmt
    return self.parse_additive_expr()
```

```
    program = parser.parse(input_text)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 30, in parse
    return self.parse_program()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 41, in parse_program
    program.body.append(self.parse_stmt())
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 49, in parse_stmt
    return self.parse_additive_expr()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 56, in parse_additive_expr
    left = self.parse_multiplicative_expr()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 70, in parse_multiplicative_expr
    left = self.parse_factor()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 96, in parse_factor
    raise Exception(__file__, 'Cannot parse current token.', self.tk())
Exception: ('c:\\Users\\MyAcer\\Desktop\\SEM 5\\CPT316\\Assignment 1\\Final\\Lexical & Syntax (AST and Parse Tree)\\Lexical & Syntax\
\Syntax Analysis\\LexicalSyntax\\parser.py', 'Cannot parse current token.', { TokenType.IDENTIFIER, j })
```

4.2.5 Test Case 5 – Error Exception

Txt File Input: *test2_invalid.txt*

main.py × test2_invalid.txt

Lexical & Syntax > Syntax Analysis > main.py > ...

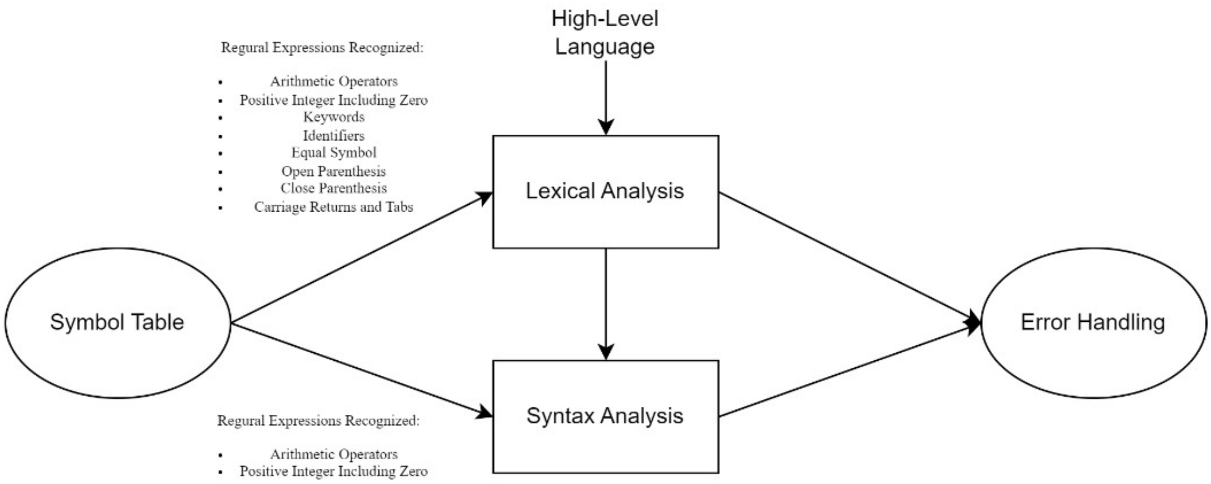
```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
    parser = LexicalSyntax.Parser()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Syntax Analysis:
>>> file
Enter the path to the .txt file: C:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexica
l & Syntax\Syntax Analysis\test2_invalid.txt
Traceback (most recent call last):
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 67, in <module>
    repl()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 42, in repl
    program = parse_input(file_content)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\main.py", line 11, in parse_input
    program = parser.parse(input_text)
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 30, in parse
    return self.parse_program()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 41, in parse_program
    program.body.append(self.parse_stmt())
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 49, in parse_stmt
    return self.parse_additive_expr()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 56, in parse_additive_expr
    left = self.parse_multiplicative_expr()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 70, in parse_multiplicative_expr
    left = self.parse_factor()
  File "c:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexical & Syntax\Syntax Analysi
s\LexicalSyntax\parser.py", line 96, in parse_factor
    raise Exception(__file__, 'Cannot parse current token.', self.tk())
Exception: ('c:\\Users\\MyAcer\\Desktop\\SEM 5\\CPT316\\Assignment 1\\Final\\Lexical & Syntax (AST and Parse Tree)\\Lexical & Syntax\
\Syntax Analysis\\LexicalSyntax\\parser.py', 'Cannot parse current token.', { TokenType.IDENTIFIER, Hello })
```

5.0 Compiler Design



6.0 Code Demonstration

lexer.py (in lexical analysis)

```
from enum import Enum, auto
import re

# Different token types for Enum
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    STRING_LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    SPACE = auto()
    EOF = auto()

# Regular expressions for each type of token
TOKEN_REGEX = {
    TokenType.NUMBER: r'[1-9]+[0-9]*|0',
    TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
    TokenType.IDENTIFIER: r'[a-zA-Z_]+[a-zA-Z_0-9]*',
    TokenType.STRING_LITERAL: r'".*?"',
    TokenType.OPERATOR: r'[+\-*/%]',
    TokenType.EQUALS: r'=',
    TokenType.OPEN_PAREN: r'\(',
    TokenType.CLOSE_PAREN: r'\)',
    TokenType.SPACE: r'[ \t\r]',
}

# Class 'Token' represent a token with their type and value
class Token:
    def __init__(self, type: TokenType, value: str) -> None:
        self.type: TokenType = type
        self.value: str = value

    def __repr__(self):
        return f'{{ {self.type}, {self.value} }}'
```

```python
# Function call to tokenize input code
def tokenize(code: str) -> list[Token]:
    tokens: list[Token] = []
    match = None

    # Continue processing the code until it's empty
    while code:
        # Iterate through token types and try to match the code with patterns
        for tokentype, pattern in TOKEN_REGEX.items():
            match = re.match(pattern, code)

            if match:
                value = match.group()
                code = code[len(value):]

                # Skip the SPACE tokens
                if tokentype == TokenType.SPACE:
                    continue
                # For STRING_LITERAL tokens, remove the quotes and add to tokens
                elif tokentype == TokenType.STRING_LITERAL:
                    tokens.append(Token(tokentype, value[1:-1]))
                # For KEYWORD tokens, add to tokens
                elif tokentype == TokenType.KEYWORD:
                    tokens.append(Token(tokentype, value))
                else:
                    # For other tokens not mentioned, add to tokens
                    tokens.append(Token(tokentype, value))
                break

        # If none of the match is found, raise an exception
        if match is None:
            raise Exception(__file__, 'Not valid character: ', code[0])

    return tokens
```

```python
# Main loop for lexical analysis
print('Lexical Analysis:')
while True:
    code = input('>>> ')
    tokens = tokenize(code)
    for token in tokens:
        print(token)
```

An essential lexical analyser is included in the code, which tokenizes input code using given token types and regular expressions. The analyser constantly seeks user input and displays the produced tokens.

main.py

```python
# Import some necessary modules
import LexicalSyntax
import sys
import os

#This is the function to parse user input or file content
def parse_input(input_text):
    # Create a parser object named parser from LexicalSyntax module
    parser = LexicalSyntax.Parser()
    # Parse the input text using the parser
    program = parser.parse(input_text)

    # Create a parser_pt object named parser from LexicalSyntax module
    parser_pt = LexicalSyntax.Parser_PT()
    # Parse the input text using the parser
    program_pt = parser_pt.parse(input_text)

    result = "Parse Tree:\n" + str(program) + "\n\nAbstract Syntax Tree:\n" + str(program_pt)

    # Return the parsed program
    return result

# This is the Read-Eval-Print Loop (REPL) function
def repl():
    print('Syntax Analysis:')

    # Accepting user input continuous until the 'exit' is entered
    while True:
        user_input = input('>>> ')
        # Check if the user wants to exit the loop or not
        if user_input.lower() == 'exit':
            break
        # Check if the user wants to input a file
        elif user_input.lower() == 'file':
            # Get the file path from the user
            file_path = input('Enter the path to the .txt file: ')
            try:
```

```python
            try:
                # Try to open and read the file
                with open(file_path, 'r') as file:
                    file_content = file.read()
                    # Parse the content of the file
                    program = parse_input(file_content)
                    print(program)
            except FileNotFoundError:
                print("File not found. Please enter a valid file path.")
        else:
            # Parse the user input
            program = parse_input(user_input)
            print(program)

if __name__ == '__main__':
    # Print the current working directory
    print("Current working directory:", os.getcwd())
    # Check if a command-line argument is provided and it is a .txt file
    if len(sys.argv) > 1 and sys.argv[1].endswith('.txt'):
        try:
            # Try to open and read the file specified in the command-line argument
            with open(sys.argv[1], 'r') as file:
                file_content = file.read()
                # Parse the content of the file
                program = parse_input(file_content)
                print(program)
        except FileNotFoundError:
            print("File not found. Please enter a valid file path.")
    else:
        # If no command is provided, enter the REPL
        repl()
```

The script above acts as a basics compiler for the python programming language, allowing users to enter code interactively or load code from files for lexical and syntax analysis.

parser.py

```python
from .lexer import Token, TokenType, tokenize
from .ast import BinaryExpr, Factor, NumberFactor, Program, Stmt
from .pt import BinaryExpr_PT, Factor_PT, NumberFactor_PT, Program_PT, Stmt_PT

# AST Parser
class Parser:
    def __init__(self) -> None:
        pass

    def tk(self) -> Token:
        return self.tokens[0]

    def eat(self) -> Token:
        """
        Return the current token and move to the next token
        """
        return self.tokens.pop(0)

    def expect(self, tokentype: TokenType) -> Token:
        if self.tk().type == tokentype:
            return self.eat()

        raise Exception(__file__, 'Token expected ', tokentype.name, ' not found. Got ', self.tk(), ' instead.')

    def parse(self, code: str) -> Program:
        # Tokenize the input code
        self.tokens = tokenize(code)

        # Start parsing the program
        return self.parse_program()

    def parse_program(self) -> Program:
        """
        Program -> Stmt*
        """
        # Create an object called program to store the parsed statements
        program = Program()

        # Parse the statements until the end of the input
        while self.tk().type != TokenType.EOF:
            program.body.append(self.parse_stmt())
        return program

    def parse_stmt(self) -> Stmt:
        """
        Stmt -> AdditiveExpr
        """
        # Parse an additive expression as a statement
        return self.parse_additive_expr()

    def parse_additive_expr(self) -> Stmt:
        """
        AdditiveExpr -> MultiplicativeExpr (( PLUS | MINUS ) MultiplicativeExpr)*
        """
        # Parse the first multiplicative expression
        left = self.parse_multiplicative_expr()

        # Continue parsing as long as there are additive operators such as plus or minus
        while self.tk().value == '+' or self.tk().value == '-':
            operator = self.eat().value
            right = self.parse_multiplicative_expr()
            left = BinaryExpr(left, operator, right)
        return left

    def parse_multiplicative_expr(self) -> Stmt:
        """
        MultiplicativeExpr -> Factor (( MUL | DIV ) Factor)*
        """
        # Parse the first factor
        left = self.parse_factor()

        # Continue parsing as long as there are multiplicative operators such as multiply (*) or divide (/)
        while self.tk().value == '*' or self.tk().value == '/':
            operator = self.eat().value
            right = self.parse_factor()
```

In the above code, it shows the parser accepts input code, tokenizes it with a lexer, and builds an Abstract Syntax Tree (AST) based on the grammar rules defined. The structure of the parsed code is represented by the AST for subsequent processing or interpretation.

lexer.py (in syntax analysis)

```
from enum import Enum, auto
import re

# Different token types
class TokenType(Enum):
    NUMBER = auto()
    KEYWORD = auto()
    IDENTIFIER = auto()
    LITERAL = auto()
    OPERATOR = auto()
    EQUALS = auto()
    OPEN_PAREN = auto()
    CLOSE_PAREN = auto()
    SPACE = auto()
    EOF = auto()

# Regular expressions for token patterns
TOKEN_REGEX = {
    TokenType.NUMBER: r'[1-9]+[0-9]*|0',
    TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
    TokenType.IDENTIFIER: r'[a-zA-Z_]+[a-zA-Z_0-9]*',
    TokenType.LITERAL: r'".*?"',
    TokenType.OPERATOR: r'[+\-*/%]',
    TokenType.EQUALS: r'=',
    TokenType.OPEN_PAREN: r'\(',
    TokenType.CLOSE_PAREN: r'\)',
    TokenType.SPACE: r'[ \t\r\n]',
}

# Token class representing a token with type and value
class Token:
    def __init__(self, type: TokenType, value: str) -> None:
        self.type: TokenType = type
        self.value: str = value
    def __repr__(self):
        return f'{{ {self.type}, {self.value} }}'
```

```python
# Function to tokenize the input code
def tokenize(code: str) -> list[Token]:
    tokens: list[Token] = []
    match = None

    # Continue processing the code until it is empty
    while code:
        # Try to match the code for each token pattern
        for tokentype, pattern in TOKEN_REGEX.items():
            match = re.match(pattern, code)

            if match:
                value = match.group()
                code = code[len(value):]

                # Skip for the SPACE tokens
                if tokentype == TokenType.SPACE:
                    continue
                # For LITERAL tokens, remove the quotes then add to tokens
                elif tokentype == TokenType.LITERAL:
                    tokens.append(Token(tokentype, value[1:-1]))
                # For KEYWORD tokens, add to tokens
                elif tokentype == TokenType.KEYWORD:
                    tokens.append(Token(tokentype, value))
                else:
                    # For other tokens not mention, add to tokens
                    tokens.append(Token(tokentype, value))
                break

        # If none of match is found, raise an exception
        if match is None:
            raise Exception(__file__, 'Not valid character: ', code[0])

    # Add EOF token to show that it is the end of tokens
    tokens.append(Token(TokenType.EOF, 'EOF'))
    return tokens
```

The lexer will use the standard regular expression or common language elements such as integers, keywords, identifiers, literals, operators and brackets to parse input code into a set of tokens which is each identifiable by its type and value.

grammar.txt

```
# Program consists of zero or more statements
Program: Stmt*

# Statement is defined as an additive expression
Stmt: AdditiveExpr

# Additive expression is composed of multiplicative expressions
# with optional + or - operators in between
AdditiveExpr: MultiplicativeExpr((PLUS | MINUS) MultiplicativeExpr)

# Multiplicative expression is composed of factors with optional
# multiplication or division operators in between
MultiplicativeExpr: Factor((MUL | DIV) Factor)*

# Factor is defined as an integer
# or AdditiveExpr with open and close parenthesis
Factor: INTEGER
    | (Open_Paren) AdditiveExpr (Close_Paren)
```

The grammar of the language specifies an easy programming language for programs that are made up of statement, and they express different types of expressions such as addition, subtraction, multiplication, division, arithmetic expressions, as well as integer literals.

ast.py

```python
# Binary expressions for class representation
class BinaryExpr(Stmt):
    def __init__(self, left: Stmt, operator: str, right: Stmt) -> None:
        self.type: NodeType = NodeType.BINARY_EXPR
        self.left: Stmt = left
        self.operator: str = operator
        self.right: Stmt = right

    def __repr__(self, indent) -> str:
        res = indent * '\t' + '{'
        res += '\n' + (indent + 1) * '\t' + self.type.name
        res += '\n' + self.left.__repr__(indent + 1)
        res += '\n' + (indent + 1) * '\t' + self.operator
        res += '\n' + self.right.__repr__(indent + 1)
        res += '\n' + indent * '\t' + '}'
        return res

# Base class for representing factor
class Factor(Stmt):
    def __init__(self) -> None:
        self.type: NodeType = NodeType.FACTOR

# Class that representing number factors
class NumberFactor(Factor):
    def __init__(self, value: int) -> None:
        self.type: NodeType = NodeType.NUMBER_FACTOR
        self.value: int = value

    def __repr__(self, indent) -> str:
        return indent * '\t' + f'{{ {self.type.name}, {self.value} }}'
```

The class hierarchy representing the structure of a programming language's Abstract Syntax Tree (AST) is shown in this code. The AST contains a program, statement, binary expression and factor as well as the numerical value type. To display an overall overview of the AST hierarchy, these classes string representations are created.

pt.py

```python
from enum import Enum, auto

# Parse Tree
# Different node types for Enum
class NodeType(Enum):
    PROGRAM = auto()
    STMT = auto()
    FACTOR = auto()
    BINARY_EXPR = auto()
    NUMBER_FACTOR = auto()

# Representation of class for the overall program
class Program_PT:
    def __init__(self) -> None:
        self.type: NodeType = NodeType.PROGRAM
        self.body: list[Stmt_PT] = []

    def __repr__(self) -> str:
        res = '{'
        res += '\n\t'

        # Represent each statement in the program
        for stmt in self.body:
            res += '\n' + stmt.__repr__(1)

        res += '\n}'
        return res

# Base class for statements
class Stmt_PT:
    def __init__(self) -> None:
        self.type: NodeType = NodeType.STMT

    def __repr__(self, indent) -> str:
        res = indent * '\t'
        return res


# Binary expressions for class representation
class BinaryExpr_PT(Stmt_PT):
    def __init__(self, left: Stmt_PT, operator: str, right: Stmt_PT) -> None:
        self.type: NodeType = NodeType.BINARY_EXPR
        self.left: Stmt_PT = left
        self.operator: str = operator
        self.right: Stmt_PT = right

    def __repr__(self, indent) -> str:
        res = indent * '\t' + '{'
        res += '\n' + (indent + 1) * '\t'
        res += '\n' + self.left.__repr__(indent + 1)
        res += '\n' + (indent + 1) * '\t' + self.operator
        res += '\n' + self.right.__repr__(indent + 1)
        res += '\n' + indent * '\t' + '}'
        return res

# Base class for representing factor
class Factor_PT(Stmt_PT):
    def __init__(self) -> None:
        self.type: NodeType = NodeType.FACTOR

# Class that representing number factors
class NumberFactor_PT(Factor_PT):
    def __init__(self, value: int) -> None:
        self.type: NodeType = NodeType.NUMBER_FACTOR
        self.value: int = value

    def __repr__(self, indent) -> str:
        return indent * '\t' + f'{{{self.value}}}'
```

init.py

```
# Import the Parser class from the local 'parser' module
from .parser import Parser, Parser_PT
```

This file initializes the package and gives the parser module access to the Parser class.

test.txt

```
Syntax Analysis > test.txt
1    2 + 5 * 3
2    4 * 5
3    (1 + 2) * 3
```

A sample of input code arithmetic operations using a text file for syntax analysis according to the defined regular expression in our compiler.

test2_invalid.txt

```
Syntax Analysis > test2_invalid.txt
1    Hello = 2 + 3 * 6
```

A sample of invalid input code using a text file for syntax analysis according to the defined regular expression in our compiler.

7.0 Design Choice and Implementation

The lexical and syntactic analysis stages of the compiler are implemented in this study using Python as our preferred language. Python was chosen because of its special blend of readability, simplicity, and flexibility. This fits in well with our objective of thoroughly demonstrating the nuances of these important compiler stages [11]. Because Python is an interpreted high-level programming language, it can be executed line by line with the aid of an interpreter, which lets us monitor and work with the code closely during compilation [10]. Furthermore, Python comes with an integrated compiler that can convert Python code into bytecode, which is the intermediate form that the interpreter runs. Lexical and syntactic analysis's core duties are embedded in this compilation process.

Beyond the previously described benefits centred around the user, there are additional benefits to implementing these phases in Python. Although the interpreted structure of Python does facilitate user-friendly interactions, our main goal is to exploit the built-in features of Python to accurately represent the subtleties of lexical and syntax analysis. Early error detection during the development phase is made easier by the Python compiler [10]. The compiler runs checks on the code during compilation to make sure it is correct and can run successfully. This entails identifying additional programming faults, adhering to naming rules, and checking for syntax errors. This comprehensive pre-execution validation saves a significant amount of time and effort at the debugging stage, which leads to a more efficient development approach. [10]

Performance gains are also obtained by compiling Python code into bytecode. Because pre-compiled bytecode doesn't require parsing and compilation every time the code executes, it can be executed faster [11]. This feature not only speeds up programme execution but also frees us from the performance constraints that come with interpreting code in real-time, allowing us to concentrate on the in-depth analysis of the lexical and syntactic analysis stages. Our program greatly increases the ease of use by offering two methods to input syntax analysis: by importing a text file from the local directory or by using user input at the terminal.

In conclusion, we chose Python as the implementation language because it strategically fits with our goal of delving deeply into the nuances of compiler phases. This decision was made based on more than just user benefits. Because of Python's characteristics, we can create a compiler that not only meets user needs but also provides an illuminating and transparent depiction of lexical and syntax analysis, revealing the inner workings of these crucial compiler phases.

8.0 Reference

[1] Mastery, T. (2023) Python compiler: Understanding and an in-depth look, DotCom Magazine-Influencers And Entrepreneurs Making News. Available at: https://dotcommagazine.com/2023/04/python-compiler-understanding-and-an-in-depth-look/

[2] Spivak, R. (2015) Let’s build a simple interpreter. part 1., Ruslan’s Blog. Available at:https://ruslanspivak.com/lsbasi-part1

[3] Nyakundi, H. (2021) How to write a good README file for your github project, freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/how-to-write-a-good-readme-file

[4] Gupta, A. (2023) 15+ popular Python IDES in 2023: Choosing the best one, Simplilearn.com. Available at: https://www.simplilearn.com/tutorials/python-tutorial/python-ide

[5] The MIT License (2023) Open Source Initiative. Available at: https://opensource.org/license/mit

[6] Aho, A. V., Sethi, R., Lam, M. S., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools. Pearson Education, Addison-Wesley, Second Edition. https://github.com/qshadun/books/blob/master/Compilers%20Principles%20Techniques%20and%20Tools%20(2nd%20Edition)%20.pdf

[7] Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler. Morgan Kaufmann Publishers. https://dl.acm.org/doi/pdf/10.5555/2737838

[8] Kemeny, J. G., & Kurtz, T. E. (1971). Basic programming. Wiley.

[9] Compiler design. Python Developer’s Guide. (n.d.). https://devguide.python.org/internals/compiler/

[10] Lutz, M. (2013). Learning Python, 5th Edition. O'Reilly Media. https://github.com/Quyaz/books/blob/master/Learning%20Python%2C%205th%20Edition.pdf

[11] Wang, J. (2016). Handbook of Finite State based Models and Applications. In Chapman and Hall/CRC eBooks. https://doi.org/10.1201/b13055

[12] Basic Syntax | Markdown Guide. (n.d.). Www.markdownguide.org. Retrieved November 16, 2023, from https://www.markdownguide.org/basic-syntax/#code

9.0 Appendix

Division of Task

Task	Member-in-Charge
Introduction to Compiler	Khausaalyaah Sinathurai
Introduction to BASIC Language	Khausaalyaah Sinathurai
Lexical Analysis	Chan Yee Shuen
Syntax Analysis	Soh Yen San
Test Cases	Veytri Yogan
Code Demonstration	Lim Yong Xin
Design Choices and Implementation	Khausaalyaah Sinathurai, Lim Yong Xin
Documentation Liaising	Veytri Yogan
Python Coding	All Members

Link for the README file

https://github.com/fionayongxin/cpt316.git

```
# Lexical and Syntax Analysis
<p>This assignment implements a simple programming language with lexical and syntax analysis. The analysis divides into two primary components:</p>

<ul>
  <li>Lexical Analysis (Lexer): Tokenizes the input code.</li>
  <li>Syntax Analysis (Parser): Constructs an Abstract Syntax Tree (AST) from tokenized code.</li>
</ul>

# Grammar
<p>The following grammar rules define the language syntax.</p>

  >**Program** : Stmt* <br><br>
  >**Stmt** : AdditiveExpr <br><br>
  >**AdditiveExpr** : MultiplicativeExpr((PLUS | MINUS) MultiplicativeExpr)* <br><br>
  >**MultiplicativeExpr** : Factor((MUL | DIV) Factor)* <br><br>
  >**Factor** : INTEGER <br>
  >(Open_Paren) AdditiveExpr (Close_Paren)
<p>These rules describe the structure of valid programs in the language.</p>

# Project Structure
Here are all the program files in ***Lexical Syntax*** folder

>`lexer.py`: Lexical analysis implementation. <br><br>
>`parser.py`: Syntax analysis implementation. <br><br>
>`ast.py`: Abstract Syntax Tree (AST) classes. <br><br>
>`main.py`: Main program execution. <br><br>
>`grammar.txt`: Grammar rules define for the programming language. <br><br>
>`test.txt`: Example of input file. <br><br>

# Abstract Syntax Tree (AST)
<p>The structure of the parsed code is defined in the AST classes:</p>

>**Program**: Represents the whole program that contains a list of statements. <br><br>
>**Stmt**: Represents a generic statement in the AST. <br><br>
>**BinaryExpr**: Represents a binary expression. <br><br>
>**Factor**: Represents a generic factor in the AST. <br><br>
>**NumberFactor**: A specific type of factor holding an integer value. <br><br>
>**NodeType**: Specifying the type of each AST node. <br><br>
```

```
# Example for input data
Please find the sample input file at `test.txt`

# Requirements
<ul>
  <li>Visual Studio Code (or any preferred IDE Python development environment)</li>
</ul>

# Getting Started
## Run the lexical analysis phase
>1. Open your preferred IDE that supports Python programming.
>2. Load the project by opening the project folder.
>3. Run the program by navigating to the `lexer.py` file in the project directory under ***Lexical Analysis*** folder.
>4. Execute the program `lexer.py` using your IDE.
>5. Enter the regular expression in the console.
>6. The output will be showing the different type of token based on regular expression entered.

## Run the Syntax analysis phase
### Method 1 (Load text file)
>1. Open your preferred IDE that supports Python programming.
>2. Load the project by opening the project folder.
>3. Run the program by navigating to the `main.py` file in the project directory under ***Lexical Syntax*** folder.
>4. Execute the program `main.py` using your IDE.
>5. To load the text file type in the word _file_ in the console.
>6. Copy and paste the path of the `test.txt` file in the console.
>7. The output is generated for both parse tree and Abstract Syntax Tree (AST).
>8. Enter _exit_ to end the program or file to input the path from a file.

### Method 2 (Enter Command)
>1. Open your preferred IDE that supports Python programming.
>2. Load the project by opening the project folder.
>3. Run the program by navigating to the `main.py` file in the project directory under ***Lexical Syntax*** folder.
>4. Execute the program `main.py` using your IDE.
>5. Enter basic arithmetic expressions command in the console.
>6. The output is generated for both parse tree and Abstract Syntax Tree (AST).
```

```
## Example regular expression
<ul>
  <li>5+7*10-3</li>
  <li>Aplus = 80%(3+2)+"Excellent"</li>
  <li>x = 7+10/2*(2+8)</li>
  <li>&</li>
  <li>final$</li>
  <li>Hello = “Yen San”</li>
</ul>

## Example arithmetic expressions
<ul>
  <li>5+7*10-3</li>
  <li>3+4-6*2</li>
  <li>j=3+r</li>
</ul>

# Contributors
<p>This assignment was developed by the following members</p>
<ol>
  <li>Veytri A/L Yogan</li>
  <li>Lim Yong Xin</li>
  <li>Chan Yee Shuen</li>
  <li>Soh Yen San</li>
  <li>Khausaalyaah A/P Sinathurai</li>
</ol>

# License
<p>This assignment codebase is licensed under the</p>
[MIT License](https://opensource.org/licenses/MIT/)
```