

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 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** ')'.
 It is used to match the closing parenthesis of a function call or a list.

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:

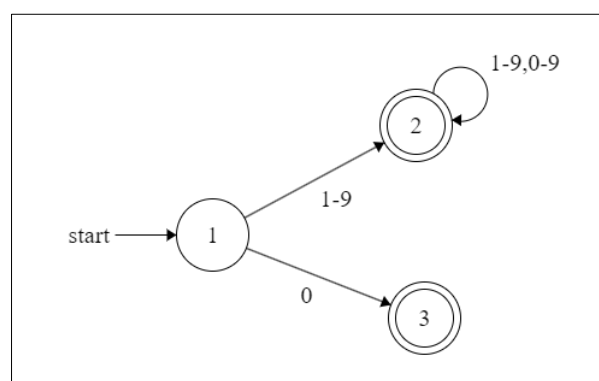


Figure 2.2.1 This figure depicts the automata which will accept positive integer including zero

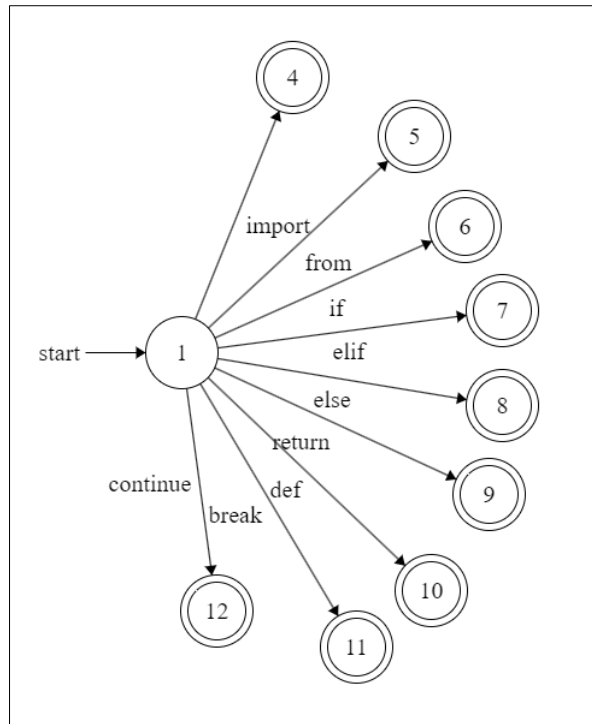


Figure 2.2.2 This figure depicts the automata which will accept keywords including "import," "from," "if," "elif," "else," "return," "def," "break," and "continue"

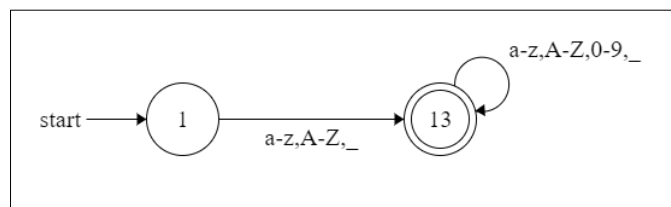


Figure 2.2.3 This figure depicts the automata which will accept identifiers that are string that begins with alphabet or underscore and ends with alphabet, integer or underscore

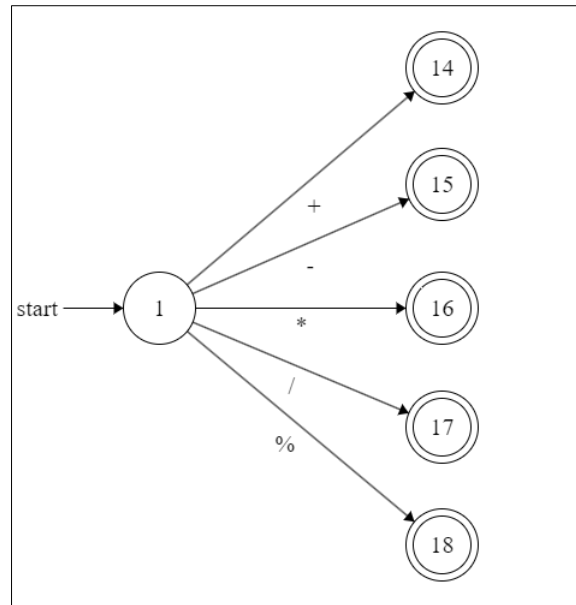


Figure 2.2.4 This figure depicts the automata which will accept arithmetic operators '+', '-', '*', '/', and '%'.

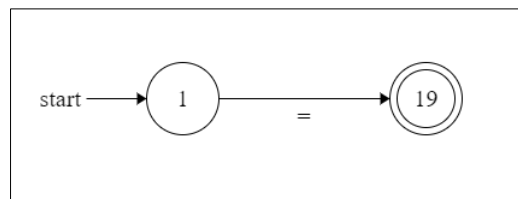


Figure 2.2.5 This figure depicts the automata which will accept equal symbol

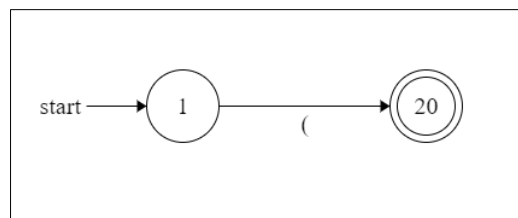


Figure 2.2.6 This figure depicts the automata which will accept open parenthesis

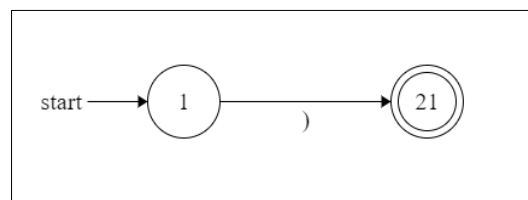


Figure 2.2.7 This figure depicts the automata which will accept close parenthesis

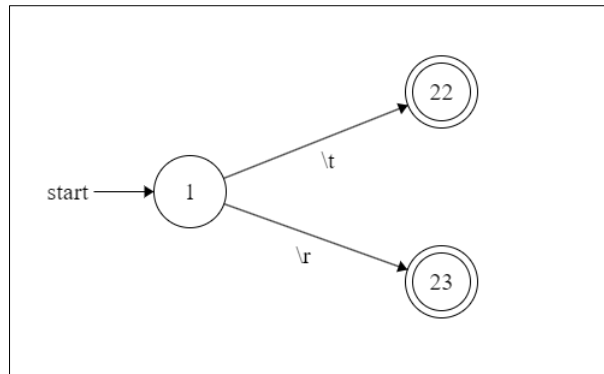


Figure 2.2.8 This figure depicts the automata which will accept carriage returns and tabs

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 --> *MultiplicativeExpr* ((*PLUS* | *MINUS*) *MultiplicativeExpr*)*

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:

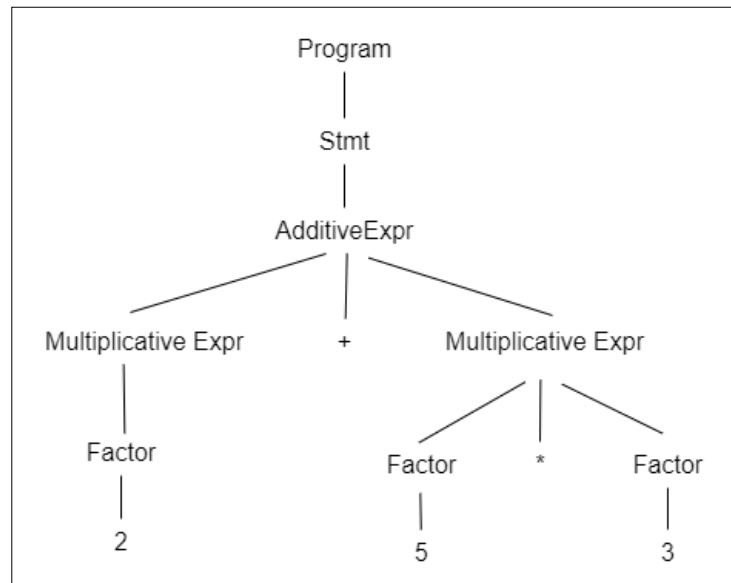


Figure 3.2.2 The figure depicts the Parse Tree generated after running the code in ast.py

Abstract Syntax Tree:

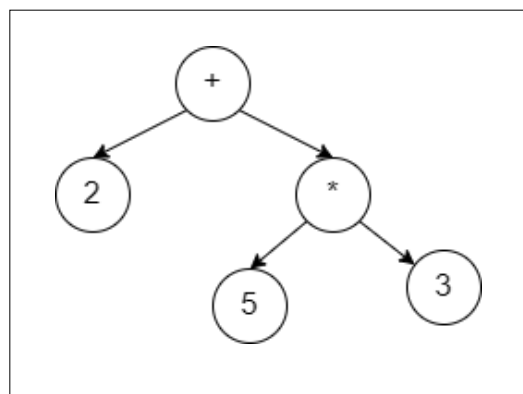


Figure 3.2.3 The figure depicts the Abstract Syntax Tree generated after running the code in ast.py

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

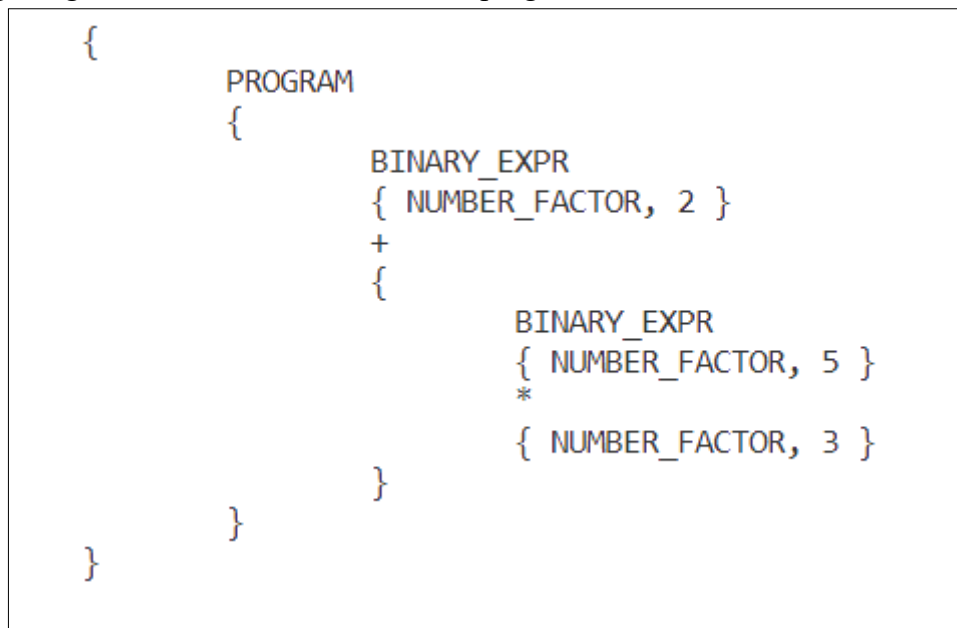


Figure 3.2.3 The figure depicts the Parse Tree generated via the compiler

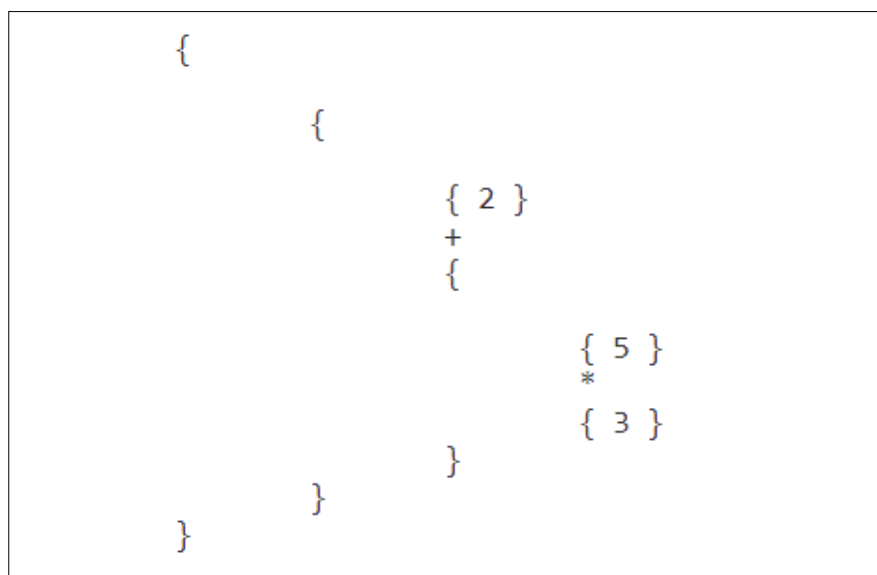


Figure 3.2.4 The figure depicts the Abstract Syntax Tree generated via the compiler

The *inner* curly bracket represents that the program will execute it first then execute the *outer* curly bracket. In parse tree, it will perform from 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.

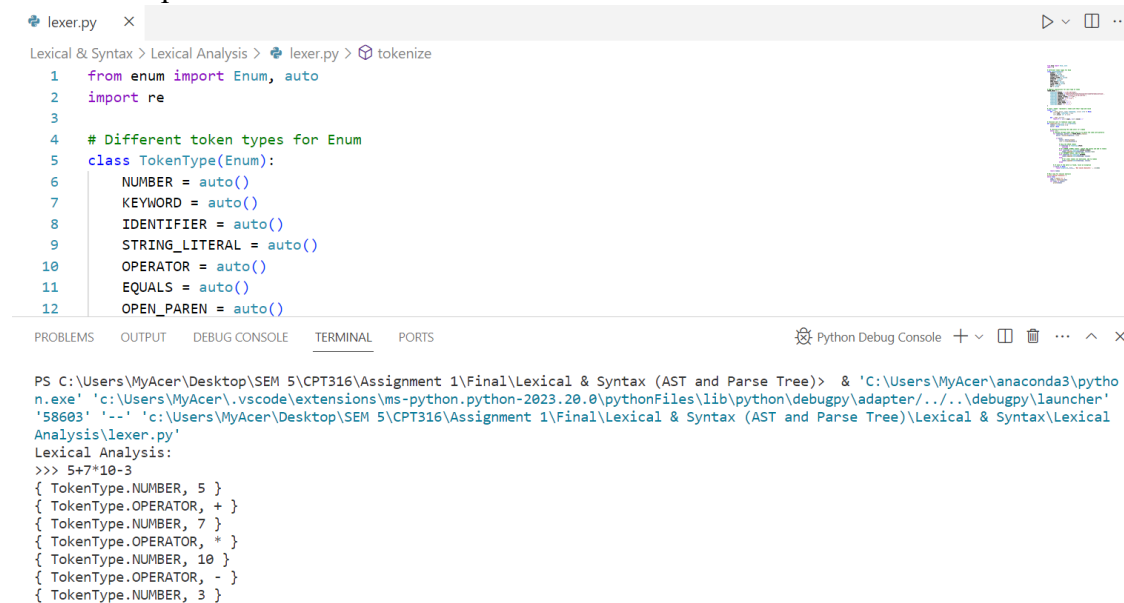
4.0 Test Cases

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: $5+7*10-3$

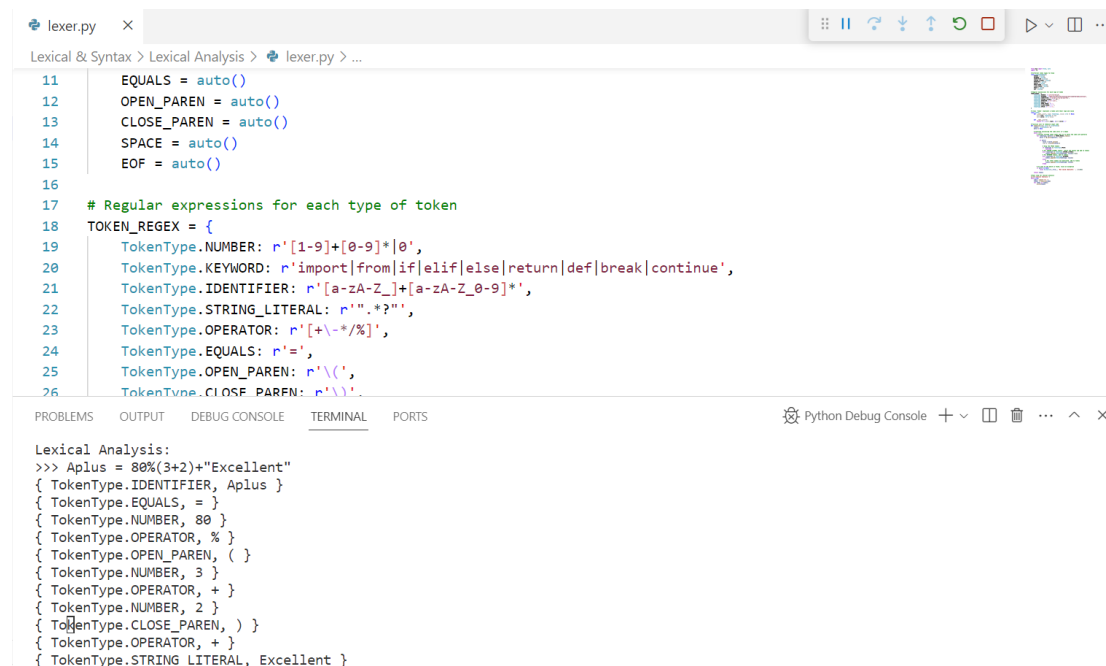


The screenshot shows the VS Code editor with the file `lexer.py` open. The code defines an enumeration `TokenType` with values `NUMBER`, `KEYWORD`, `IDENTIFIER`, `STRING_LITERAL`, `OPERATOR`, `EQUALS`, and `OPEN_PAREN`. The terminal output shows the execution of the program with the input `5+7*10-3`, resulting in a list of tokens: `{ TokenType.NUMBER, 5 }`, `{ TokenType.OPERATOR, + }`, `{ TokenType.NUMBER, 7 }`, `{ TokenType.OPERATOR, * }`, `{ TokenType.NUMBER, 10 }`, `{ TokenType.OPERATOR, - }`, and `{ TokenType.NUMBER, 3 }`.

```
lexer.py x
Lexical & Syntax > Lexical Analysis > lexer.py > tokenize
1 from enum import Enum, auto
2 import re
3
4 # Different token types for Enum
5 class TokenType(Enum):
6     NUMBER = auto()
7     KEYWORD = auto()
8     IDENTIFIER = auto()
9     STRING_LITERAL = auto()
10    OPERATOR = auto()
11    EQUALS = auto()
12    OPEN_PAREN = auto()
13
14 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: $Aplus = 80\%(3+2)+\text{"Excellent"}$

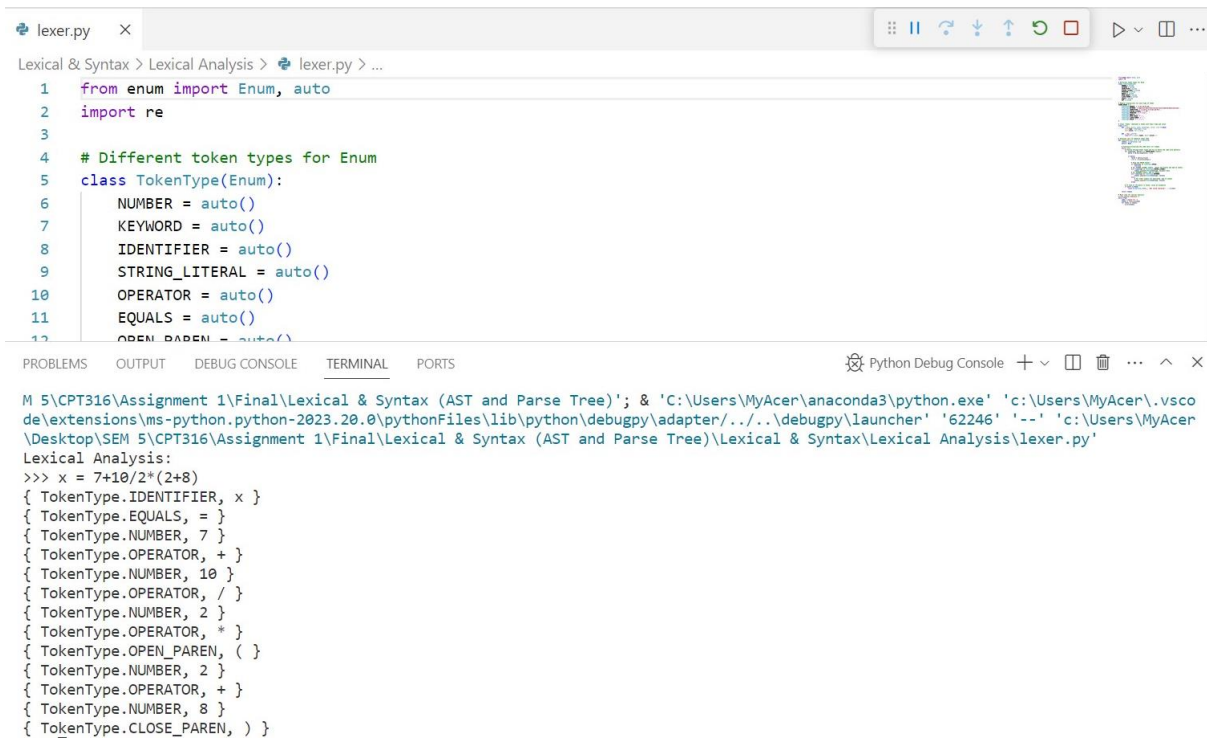


The screenshot shows the VS Code editor with the file `lexer.py` open. The code defines an enumeration `TokenType` with values `NUMBER`, `KEYWORD`, `IDENTIFIER`, `STRING_LITERAL`, `OPERATOR`, `EQUALS`, `OPEN_PAREN`, `CLOSE_PAREN`, and `SPACE`. The terminal output shows the execution of the program with the input `Aplus = 80%(3+2)+Excellent`, resulting in a list of tokens: `{ 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, + }`, and `{ TokenType.STRING_LITERAL, Excellent }`.

```
lexer.py x
Lexical & Syntax > Lexical Analysis > lexer.py > ...
11 EQUALS = auto()
12 OPEN_PAREN = auto()
13 CLOSE_PAREN = auto()
14 SPACE = auto()
15 EOF = auto()
16
17 # Regular expressions for each type of token
18 TOKEN_REGEX = {
19     TokenType.NUMBER: r'[1-9]+[0-9]*|0',
20     TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
21     TokenType.IDENTIFIER: r'[a-zA-Z_]+[a-zA-Z_0-9]*',
22     TokenType.STRING_LITERAL: r'\".*?\"',
23     TokenType.OPERATOR: r'[+|-|*|/]',
24     TokenType.EQUALS: r'=',
25     TokenType.OPEN_PAREN: r'(',
26     TokenType.CLOSE_PAREN: r')',
27 }
28
29 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

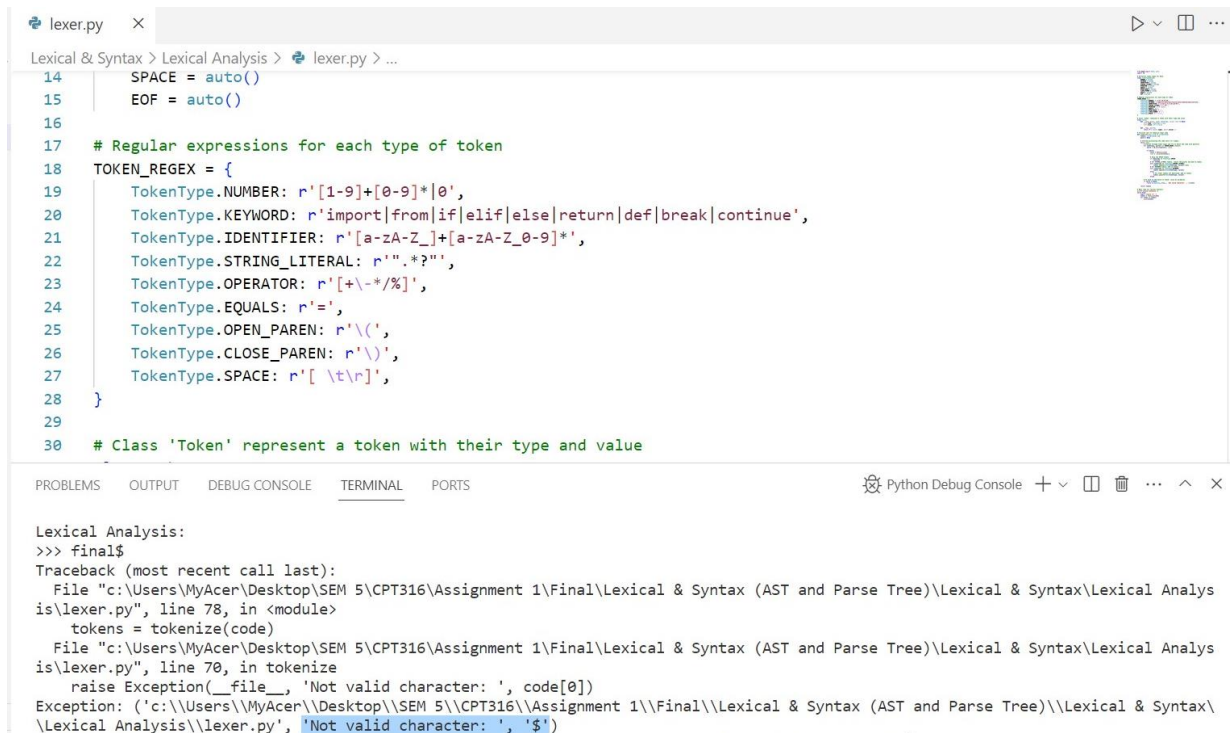
Terminal Input: $x = 7 + 10 / 2 * (2 + 8)$



```
lexer.py x
Lexical & Syntax > Lexical Analysis > lexer.py > ...
1 from enum import Enum, auto
2 import re
3
4 # Different token types for Enum
5 class TokenType(Enum):
6     NUMBER = auto()
7     KEYWORD = auto()
8     IDENTIFIER = auto()
9     STRING_LITERAL = auto()
10    OPERATOR = auto()
11    EQUALS = auto()
12    OPEN_PAREN = auto()
13    CLOSE_PAREN = auto()
14
15 # Regular expressions for each type of token
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
263
```

4.1.5 Test Case 5 – Error Exception

Terminal Input: *final\$*



```
lexer.py x
Lexical & Syntax > Lexical Analysis > lexer.py > ...
14 SPACE = auto()
15 EOF = auto()
16
17 # Regular expressions for each type of token
18 TOKEN_REGEX = {
19     TokenType.NUMBER: r'[1-9]+[0-9]*\b',
20     TokenType.KEYWORD: r'import|from|if|elif|else|return|def|break|continue',
21     TokenType.IDENTIFIER: r'[a-zA-Z_][a-zA-Z_0-9]*',
22     TokenType.STRING_LITERAL: r'\".*?\"',
23     TokenType.OPERATOR: r'[+|-|*|/]',
24     TokenType.EQUALS: r'=',
25     TokenType.OPEN_PAREN: r'(',
26     TokenType.CLOSE_PAREN: r')',
27     TokenType.SPACE: r'[\t\r]',
28 }
29
30 # Class 'Token' represent a token with their type and value

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python Debug Console + - [] ... ^ X

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 Analysis\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 Analysis\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: ', '$')
```

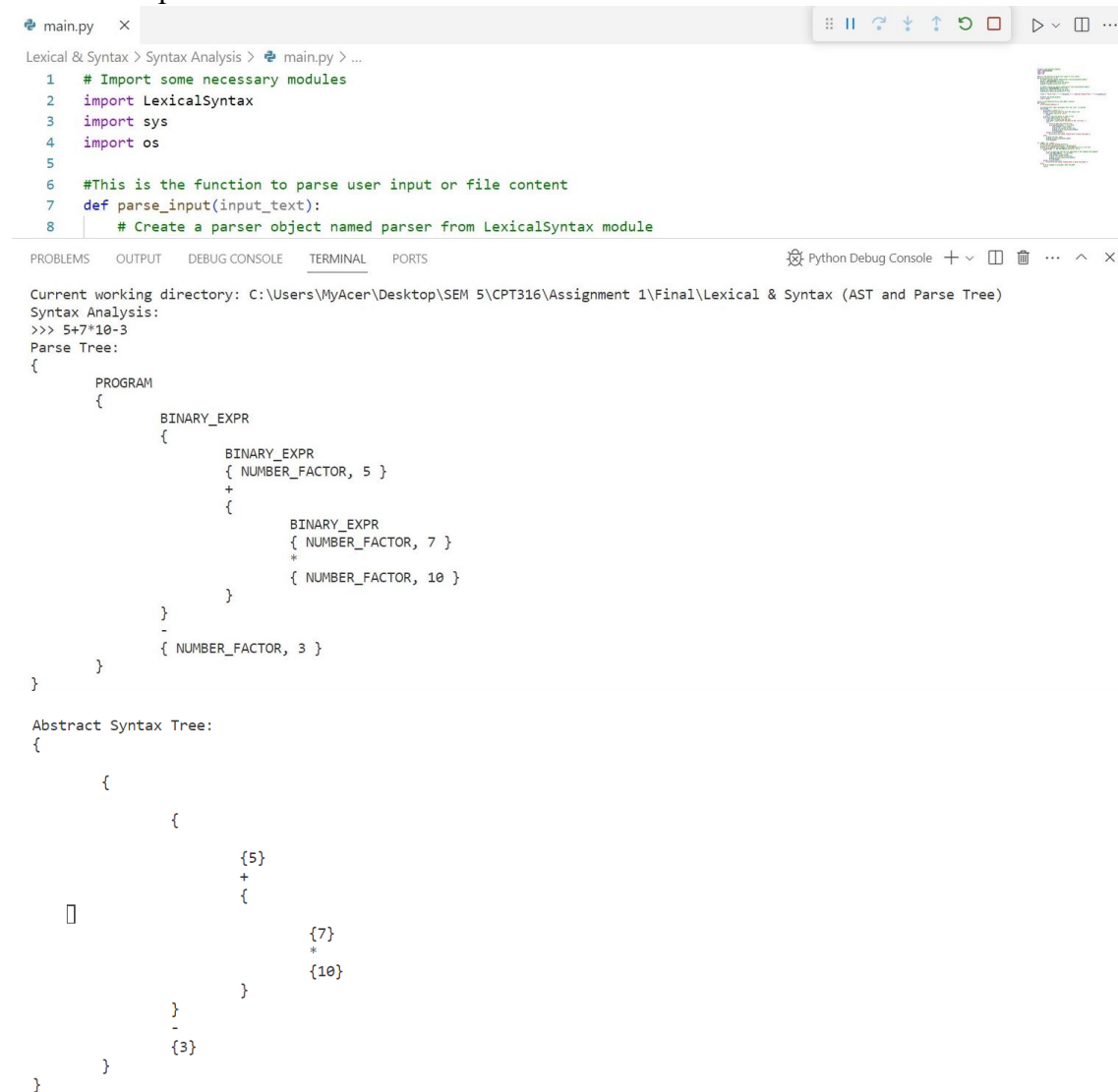
Since '\$' character is not accepted, therefore the program will raise the error exception and prompt the message to remind user that '\$' is not a valid character.

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: $5+7*10-3$



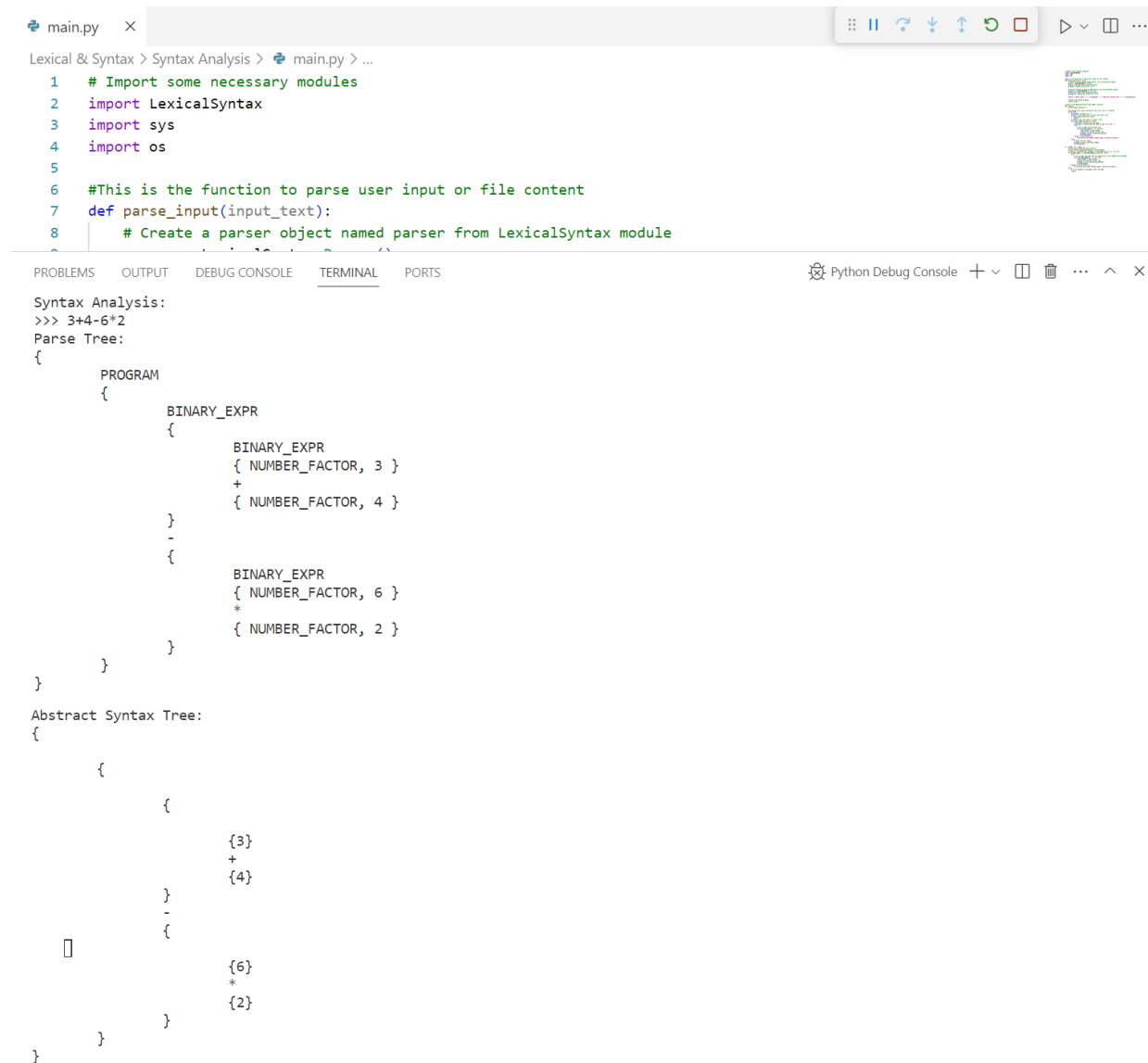
```
main.py x
Lexical & Syntax > Syntax Analysis > main.py > ...
1 # Import some necessary modules
2 import LexicalSyntax
3 import sys
4 import os
5
6 #This is the function to parse user input or file content
7 def parse_input(input_text):
8     # Create a parser object named parser from LexicalSyntax module

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 x [debug icons]
Lexical & Syntax > Syntax Analysis > main.py > ...
1 # Import some necessary modules
2 import LexicalSyntax
3 import sys
4 import os
5
6 #This is the function to parse user input or file content
7 def parse_input(input_text):
8     # Create a parser object named parser from LexicalSyntax module
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

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 x test.txt
Lexical & Syntax > Syntax Analysis > main.py > ...
1 # Import some necessary modules
2 import LexicalSyntax
3 import sys
4 import os
5
6 #This is the function to parse user input or file content
7 def parse_input(input_text):
8     # Create a parser object named parser from LexicalSyntax module
9     parser = LexicalSyntax.Parser()

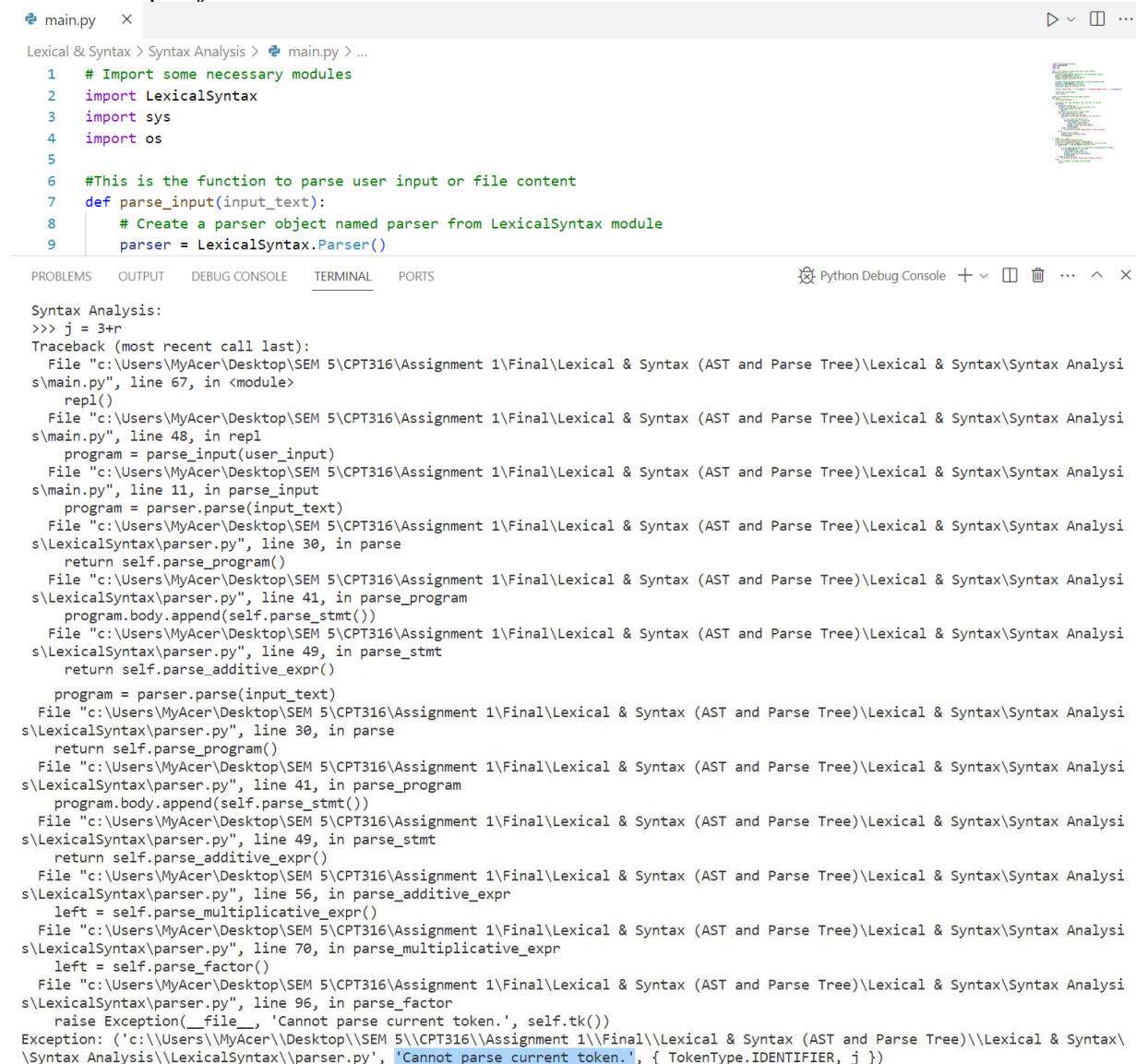
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console
Syntax Analysis:
>>> file
Enter the path to the .txt file: C:\Users\MyAcer\Desktop\SEM 5\CPT316\Assignment 1\Final\Lexical & Syntax (AST and Parse Tree)\Lexica
1 & Syntax\Syntax Analysis\test.txt

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: $j = 3+r$

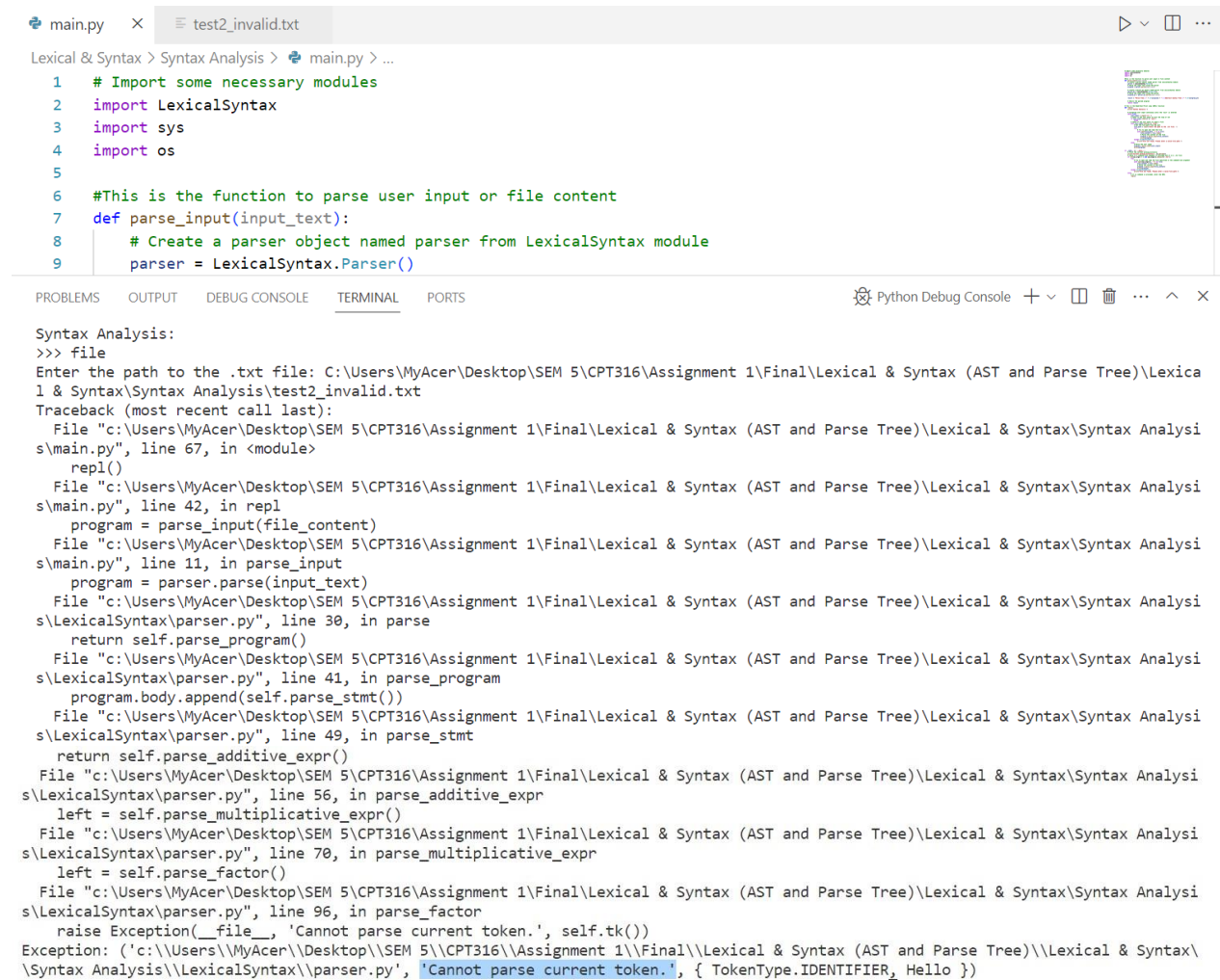


```
main.py x
Lexical & Syntax > Syntax Analysis > main.py > ...
1 # Import some necessary modules
2 import LexicalSyntax
3 import sys
4 import os
5
6 #This is the function to parse user input or file content
7 def parse_input(input_text):
8     # Create a parser object named parser from LexicalSyntax module
9     parser = LexicalSyntax.Parser()

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 Analysis\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 Analysis\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 Analysis\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 Analysis\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 Analysis\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 Analysis\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 Analysis\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 Analysis\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 Analysis\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

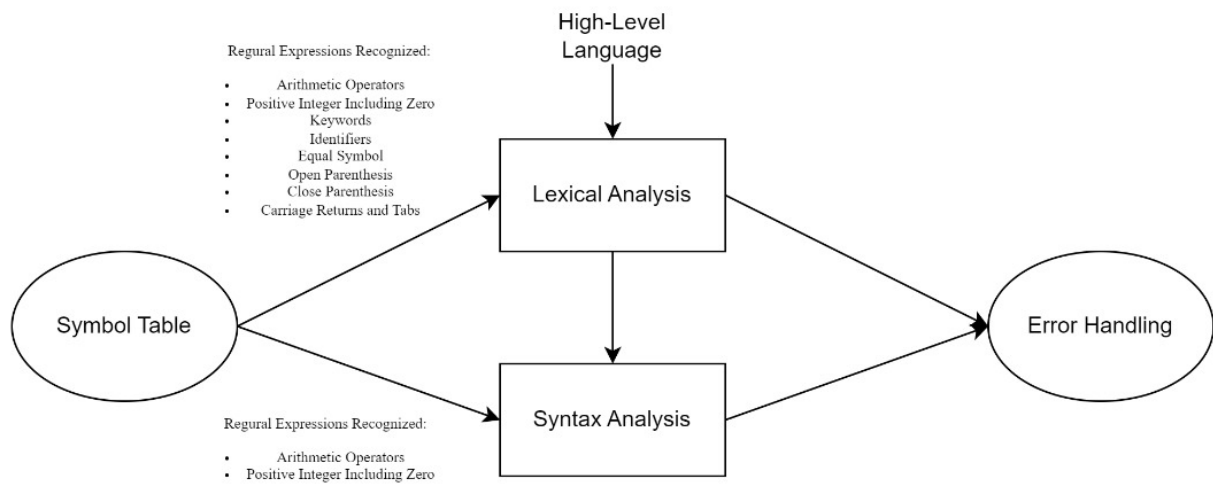
Txt File Input: *test2_invalid.txt*



```
main.py x test2_invalid.txt
Lexical & Syntax > Syntax Analysis > main.py > ...
1 # Import some necessary modules
2 import LexicalSyntax
3 import sys
4 import os
5
6 #This is the function to parse user input or file content
7 def parse_input(input_text):
8     # Create a parser object named parser from LexicalSyntax module
9     parser = LexicalSyntax.Parser()

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)\Lexical
1 & 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} }}'
```

```

# 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

# 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

```
# 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:
                # 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

```
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} }}'
```

```

# 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

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

```
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/lbasi-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](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](https://www.markdownguide.org/basic-syntax/#code). 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/fionavongxin/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/)

```