MULTIMEDIA UNIVERSITY

FACULTY OF COMPUTING AND INFORMATICS (FCI)

TCP 2451 Programming Language Translation

Mini Project

TRIMESTER 1, 2020/2021

Prepared by

Student Name	**:**	**Kathiressan A/L Sivanes**
Student ID	**:**	**1171100811**
Contact No.	**:**	**+60 12-444 3565**
E-mail Address	**:**	**1171100811@student.mmu.edu.my**
Course Name	**:**	**Bachelor of Computer Science (Hons) in Data Science**

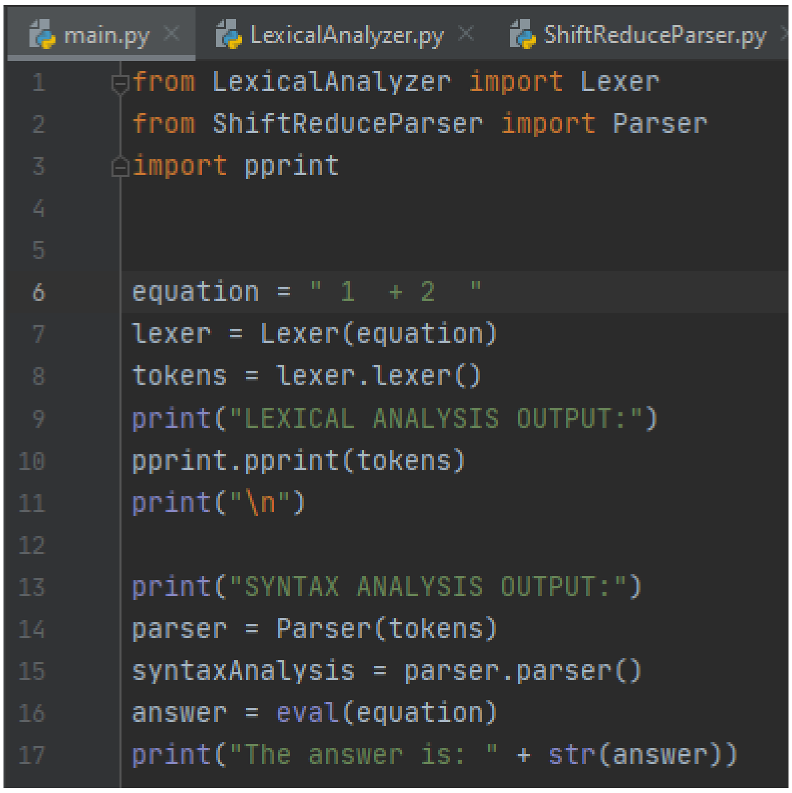
Subject Lecturer and Tutor,

Mr. Nathar Shah Bin Packier Mohammad

Table of Contents

1.0 Introduction

This mini project is completed using the Python programming language. The program is separated into three files called *main.py*, *LexicalAnalyzer.py* and *ShiftReduceParser.py*. *LexicalAnalyzer.py* and *ShiftReduceParser.py* will be further explained in following chapters. The source code for “main.py” is shown in the figure below.



```python
from LexicalAnalyzer import Lexer
from ShiftReduceParser import Parser
import pprint


equation = " 1   + 2   "
lexer = Lexer(equation)
tokens = lexer.lexer()
print("LEXICAL ANALYSIS OUTPUT:")
pprint.pprint(tokens)
print("\n")

print("SYNTAX ANALYSIS OUTPUT:")
parser = Parser(tokens)
syntaxAnalysis = parser.parser()
answer = eval(equation)
print("The answer is: " + str(answer))
```

Figure 1: *main.py* source code

Based on the Figure 1, *main.py* just calls the classes from the other two files and provides the mathematical equation to parse. It also prints the output of lexical analysis and evaluation of the equation. The *pprint* library just helps to print in nicely formatted way. “LA” stands for Lexical Analysis and “SA” stands for Syntax Analysis. These abbreviations will be used later on.

2.0 Designing a Language

The mathematical language is designed by using context-free grammar written in Backnus-Naur (BNF) form.

Start Symbol: <EXP>, which is an expression

Terminal Symbols: {+, -, *, /, (,), NUM}, where "NUM" is any number

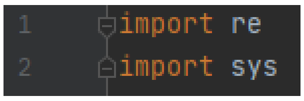
Production Rules:

```
(1) <EXP> ::= NUM
(2) <EXP> ::=  -<EXP>
(3) <EXP> ::= (<EXP>)
(4) <EXP> ::= <EXP> + <EXP>
(5) <EXP> ::= <EXP> - <EXP>
(6) <EXP> ::= <EXP> * <EXP>
(7) <EXP> ::= <EXP> / <EXP>
```

The mathematical language allows unary minus, addition, subtraction, multiplication, and division operators. The language also allows brackets to supersede other operations. The next part is lexical analysis where the given mathematical equation will go through a lexical analyser to check if all the terminal symbols are valid.

3.0 Lexical Analysis

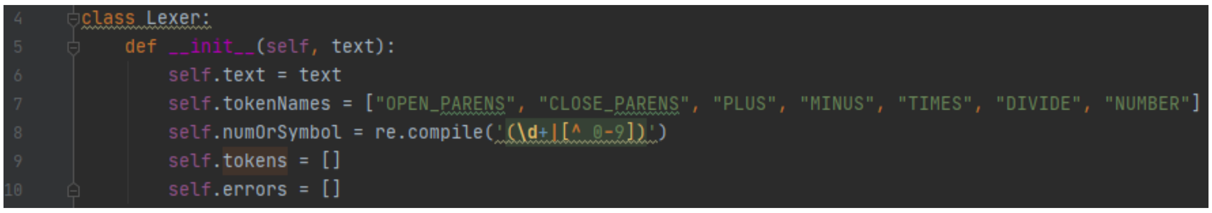
The lexical analyser source code is in the *LexicalAnalyzer.py* file. The source code is shown and explained below.



```python
import re
import sys
```

Figure 2: LA imports

Figure 2 shows all the imported libraries for the *LexicalAnalyzer.py* file. *re* is the regular expression library and *sys* library is the system-specific parameters and functions library which provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.



```python
class Lexer:
    def __init__(self, text):
        self.text = text
        self.tokenNames = ["OPEN_PARENS", "CLOSE_PARENS", "PLUS", "MINUS", "TIMES", "DIVIDE", "NUMBER"]
        self.numOrSymbol = re.compile('(\d+|[^ 0-9])')
        self.tokens = []
        self.errors = []
```

Figure 3: LA Method 1

After that, a class called *Lexer* is created as shown in Figure 3. The first method in the class is called __*init*__ which takes in parameters *self* and *text*. The *self* parameter is used to represent the instance of the class while the *text* parameter takes in any given mathematical equation.

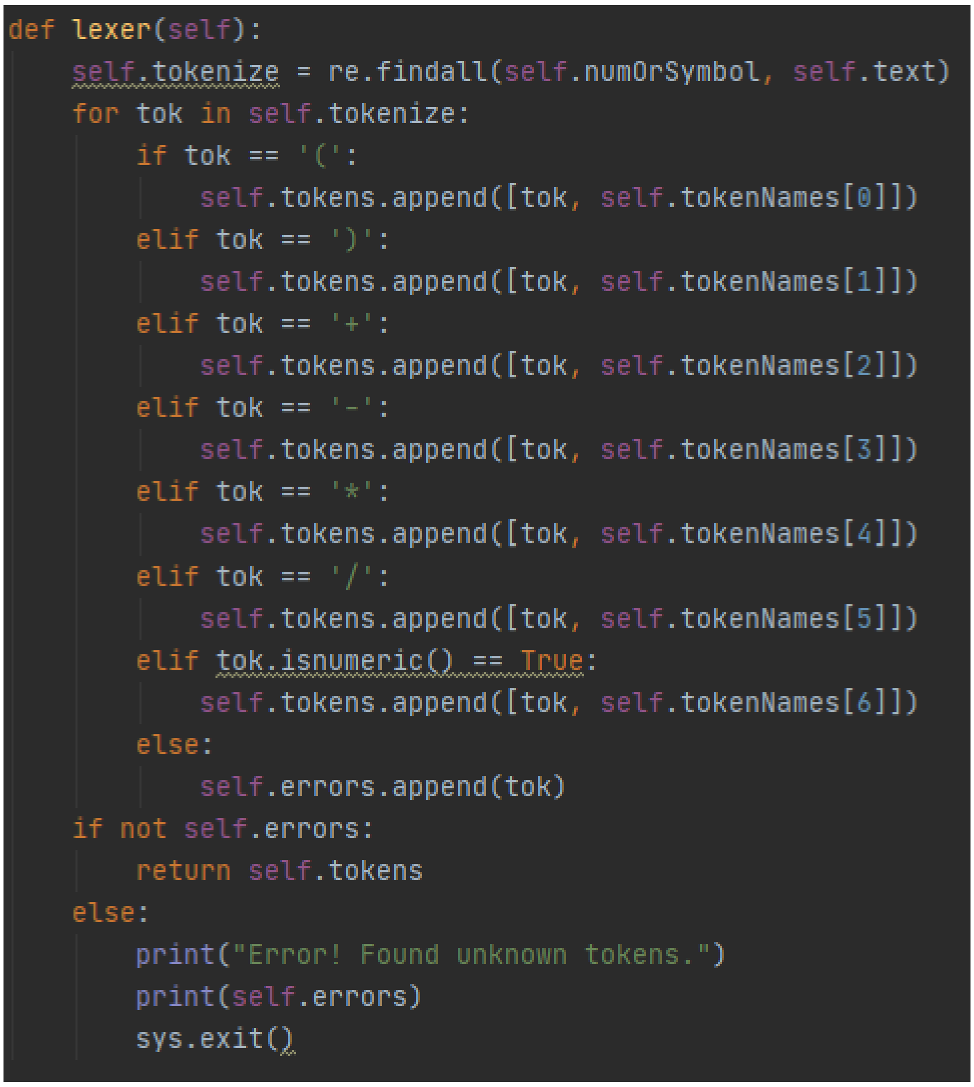
Valid tokens accepted by the program are created in *self.tokenNames* variable. The accepted tokens are shown below.

Table 1: Token Table

No	**Token Symbol**	**Token Name**
1	(	OPEN_PARANS
2	)	CLOSE_PARENS
3	+	PLUS
4	-	MINUS
6	*	TIMES
7	/	DIVIDE
8	Any Number Containing Digits 0-9	NUM

self.numOrSymbol contains the rules for the regular expression to split a mathematical equation into the stated tokens. The regular expression used is “\d+|[^ 0-9]”. The regular expression carries the following meaning:

- *\d+* means any digit which occurs one or more times
- | is a Boolean OR
- [^ 0-9] means match any character that is not a number



```python
def lexer(self):
    self.tokenize = re.findall(self.numOrSymbol, self.text)
    for tok in self.tokenize:
        if tok == '(':
            self.tokens.append([tok, self.tokenNames[0]])
        elif tok == ')':
            self.tokens.append([tok, self.tokenNames[1]])
        elif tok == '+':
            self.tokens.append([tok, self.tokenNames[2]])
        elif tok == '-':
            self.tokens.append([tok, self.tokenNames[3]])
        elif tok == '*':
            self.tokens.append([tok, self.tokenNames[4]])
        elif tok == '/':
            self.tokens.append([tok, self.tokenNames[5]])
        elif tok.isnumeric() == True:
            self.tokens.append([tok, self.tokenNames[6]])
        else:
            self.errors.append(tok)
    if not self.errors:
        return self.tokens
    else:
        print("Error! Found unknown tokens.")
        print(self.errors)
        sys.exit()
```

Figure 4: LA Method 2

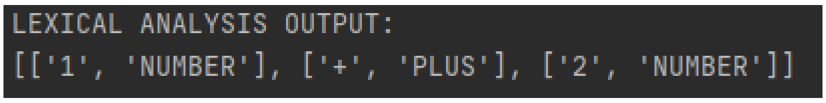
The second method in the class is called *lexer*. *self.tokenize* uses the regular expression to find all the rules in the expression and splits it up. The following for-loop loops through the split expression to check if all the symbols match any of the symbols or numeric value in Table 1. If there are any invalid symbols, they are stored in a variable called *self.errors* and the program will display all the invalid tokens before the program exits. If all the symbols are valid, the program will continue to do syntax analysis.

Examples of lexical analysis for valid tokens:

Example 1

Equation: 1 + 2

Output:



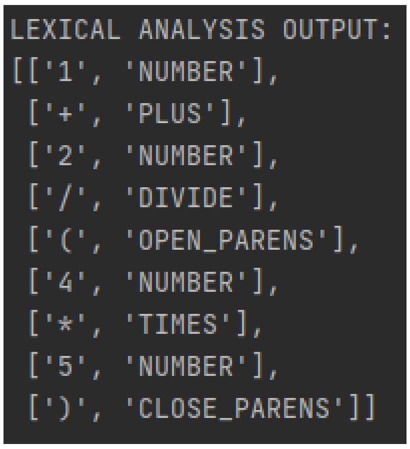
```
LEXICAL ANALYSIS OUTPUT:
[['1', 'NUMBER'], ['+', 'PLUS'], ['2', 'NUMBER']]
```

Figure 5: LA Example 1

Example 2

Equation: 1+2/(4*5)

Output:



```
LEXICAL ANALYSIS OUTPUT:
[['1', 'NUMBER'],
 ['+', 'PLUS'],
 ['2', 'NUMBER'],
 ['/', 'DIVIDE'],
 ['(', 'OPEN_PARENS'],
 ['4', 'NUMBER'],
 ['*', 'TIMES'],
 ['5', 'NUMBER'],
 [')', 'CLOSE_PARENS']]
```

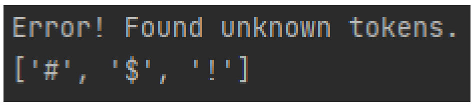
Figure 6: LA Example 2

Example of lexical analysis for invalid tokens:

Example 1

Equation: 1#+2+$!

Output:

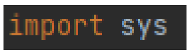


```
Error! Found unknown tokens.
['#', '$', '!']
```

Figure 7: LA Example 3

4.0 Syntax Analysis

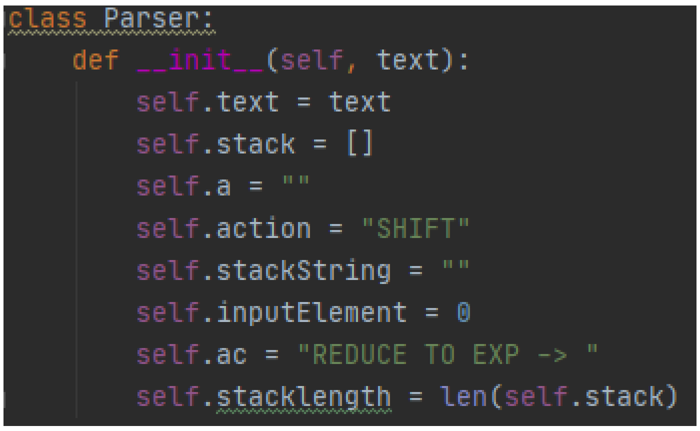
The syntax analysis source code is in the *ShiftReduceParser.py* file. The source code is shown and explained below.



```python
import sys
```

Figure 8: SA imports

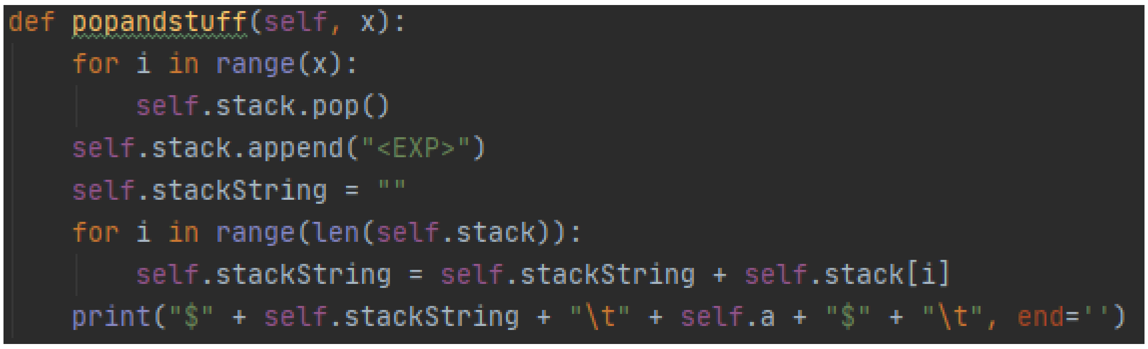
Figure 8 shows all the imported libraries for the *ShiftReduceParser.py* file. *sys* library is the system-specific parameters and functions library which provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.



```python
class Parser:
    def __init__(self, text):
        self.text = text
        self.stack = []
        self.a = ""
        self.action = "SHIFT"
        self.stackString = ""
        self.inputElement = 0
        self.ac = "REDUCE TO EXP -> "
        self.stacklength = len(self.stack)
```

Figure 9: SA Method 1

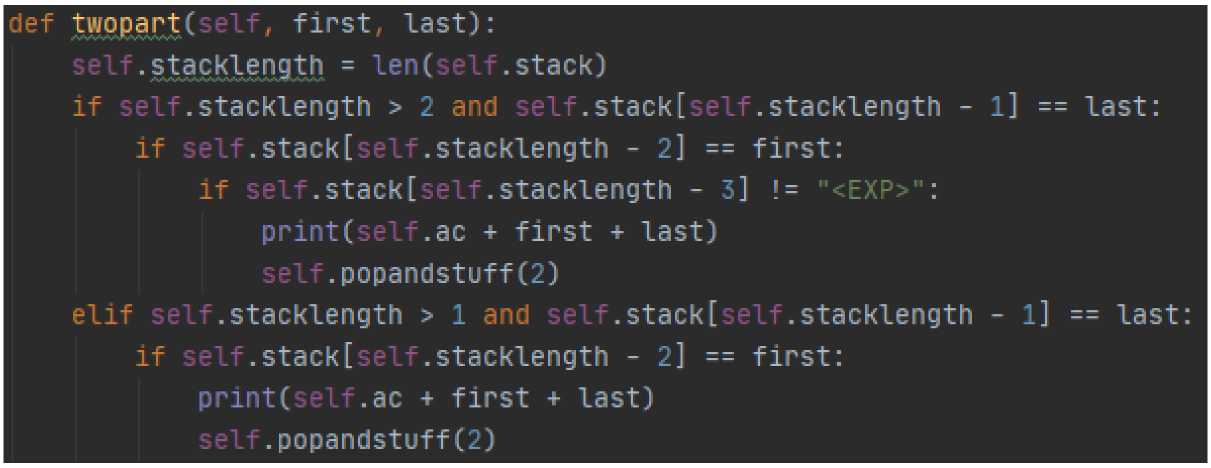
After that, a class called *Parser* is created as shown in Figure 9. The first method in the class is called *__init__* which takes in parameters *self* and *text*. The *self* parameter is used to represent the instance of the class while the *text* parameter takes in all the tokens obtained from Lexical Analysis. I created a few variables which I will use later on. *self.stack* is the stack and *self.stacklength* is the length of the stack.



```
def popandstuff(self, x):
    for i in range(x):
        self.stack.pop()
    self.stack.append("<EXP>")
    self.stackString = ""
    for i in range(len(self.stack)):
        self.stackString = self.stackString + self.stack[i]
    print("$" + self.stackString + "\t" + self.a + "$" + "\t", end='')
```

Figure 10: SA Method 2

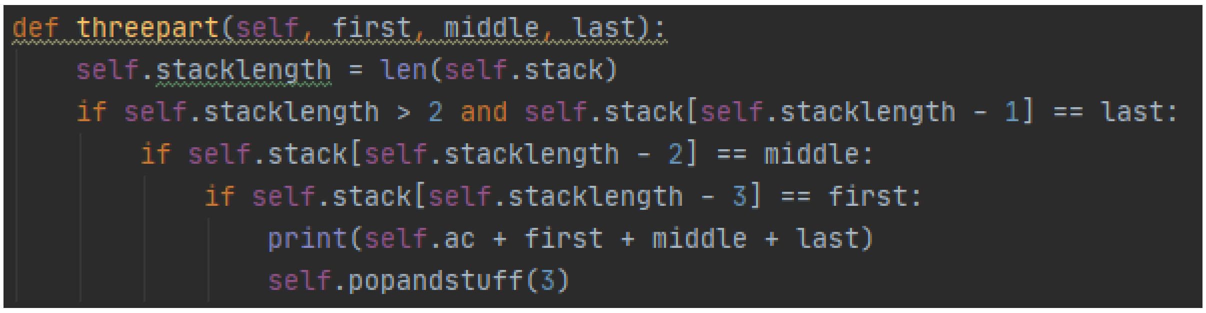
The second method in the class is called *popandstuff*. The method takes in two parameters which are *self* and *x*. *x* is any number of times to pop off the stack. Once all the items are popped from the stack, *<EXP>* is added to the stack. Next, The items in the stack, the remaining tokens in the equation and the corresponding action is printed on the screen.



```
def twopart(self, first, last):
    self.stacklength = len(self.stack)
    if self.stacklength > 2 and self.stack[self.stacklength - 1] == last:
        if self.stack[self.stacklength - 2] == first:
            if self.stack[self.stacklength - 3] != "<EXP>":
                print(self.ac + first + last)
                self.popandstuff(2)
    elif self.stacklength > 1 and self.stack[self.stacklength - 1] == last:
        if self.stack[self.stacklength - 2] == first:
            print(self.ac + first + last)
            self.popandstuff(2)
```

Figure 11: SA Method 3

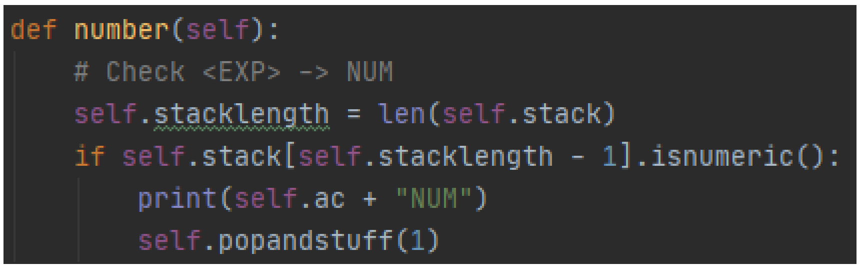
The third method is called *twopart*. The method takes in three parameters which are *self*, *first* and *last*. This method basically checks for all production rules having two parts on the right hand side. For example, *<EXP>* -> -*<EXP>*. The unary minus production rule only has two parts which are – and *<EXP>*. *first* takes in the left half and *last* takes in the right half. The stack is checked to see if there are any occurrences of *first* and *last* part together at the top of the stack. If there is an occurrence, *self.popandstuff* is called and 2 is the parameter because there is two parts to the production rule.



```python
def threepart(self, first, middle, last):
    self.stacklength = len(self.stack)
    if self.stacklength > 2 and self.stack[self.stacklength - 1] == last:
        if self.stack[self.stacklength - 2] == middle:
            if self.stack[self.stacklength - 3] == first:
                print(self.ac + first + middle + last)
                self.popandstuff(3)
```

Figure 12: SA Method 4

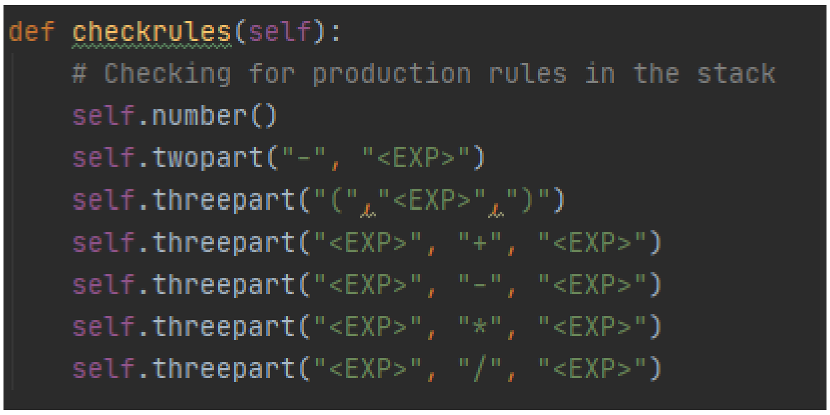
The fourth method is called *threepart*. The method takes in four parameters which are *self*, *first*, *middle*, and *last*. Similar to *twopart*, this method checks for all production rules having three parts on the right hand side. For example, *<EXP> -> (<EXP>)*. The parathesis production rule has three parts which are *(*, *<EXP>* and *)*. *First* takes in the left third, *middle* takes in the middle third and *last* takes in the right third. The stack is checked to see if there are any occurrences of *first*, *middle* and *last* together at the top of the stack. If there is an occurrence, *self.popandstuff* is called and 3 is the parameter because there are three parts to the production rule.



```python
def number(self):
    # Check <EXP> -> NUM
    self.stacklength = len(self.stack)
    if self.stack[self.stacklength - 1].isnumeric():
        print(self.ac + "NUM")
        self.popandstuff(1)
```

Figure 13: SA Method 5

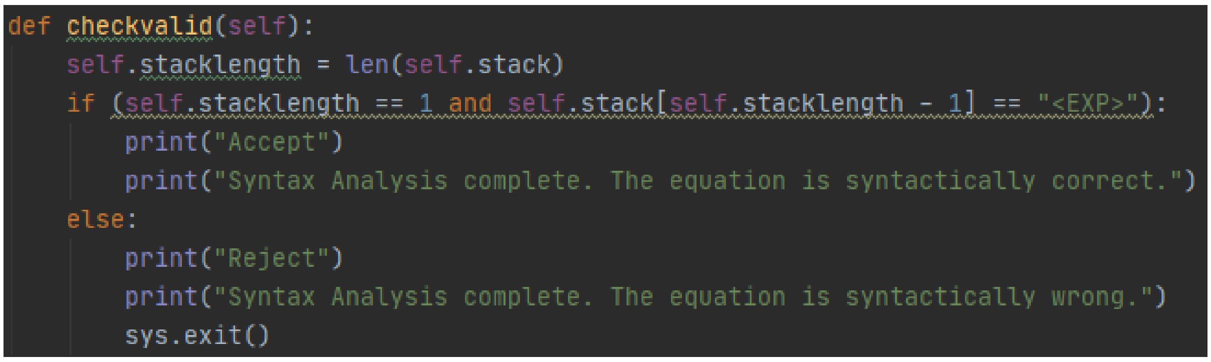
The fifth method is called *number*. The method takes in only one parameter which is *self*. This method checks for the number production rule which is *<EXP> -> NUM*. The stack is checked to see if there is any number on the top of the stack. If there is a number, *self.popandstuff* is called and 1 is the parameter because there is only one number.



```python
def checkrules(self):
    # Checking for production rules in the stack
    self.number()
    self.twopart("-", "<EXP>")
    self.threepart("(","<EXP>",")")
    self.threepart("<EXP>", "+", "<EXP>")
    self.threepart("<EXP>", "-", "<EXP>")
    self.threepart("<EXP>", "*", "<EXP>")
    self.threepart("<EXP>", "/", "<EXP>")
```

Figure 14: SA Method 6

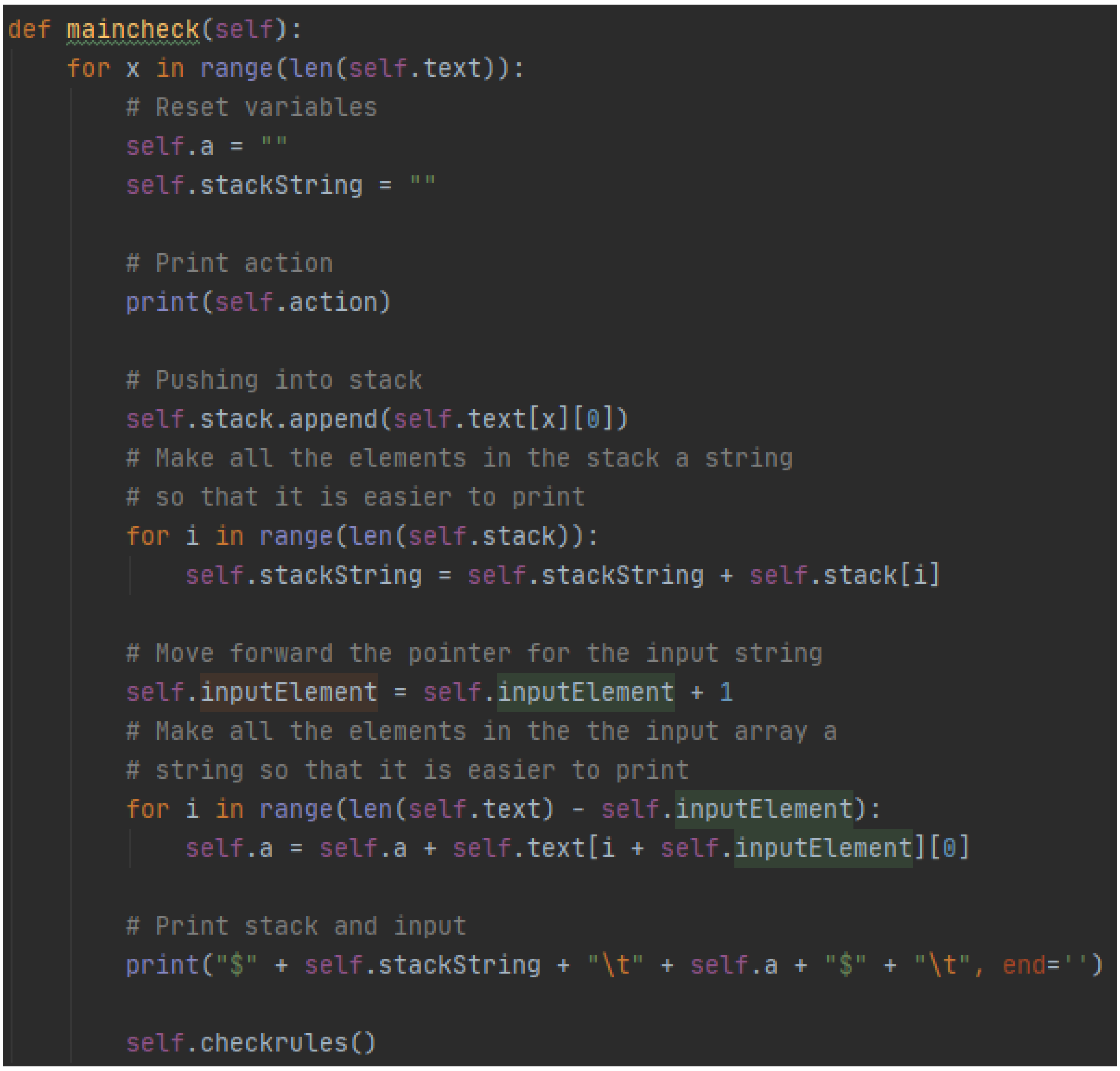
The sixth method is called *checkrules*. The method takes in only one parameter which is *self*. This method just calls the other methods which are *self.number*, *self.twopart* and *self.threepart* and adds the correct parameters for each production rule.



```python
def checkvalid(self):
    self.stacklength = len(self.stack)
    if (self.stacklength == 1 and self.stack[self.stacklength - 1] == "<EXP>"):
        print("Accept")
        print("Syntax Analysis complete. The equation is syntactically correct.")
    else:
        print("Reject")
        print("Syntax Analysis complete. The equation is syntactically wrong.")
        sys.exit()
```

Figure 15: SA Method 7

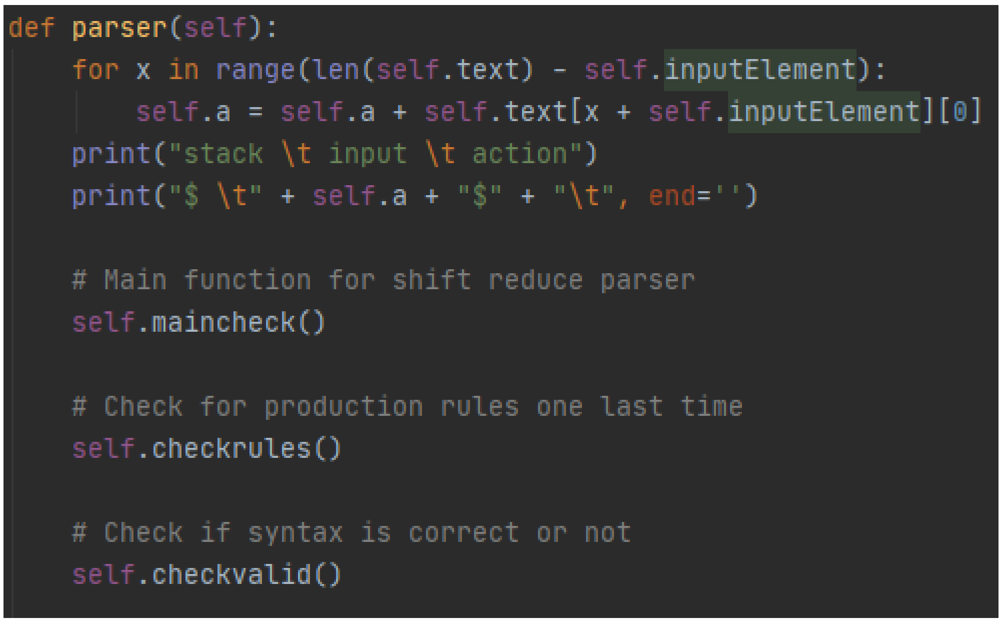
The seventh method is called *checkvalid*. This method checks if there is only 1 item in the stack and if that item is *<EXP>*. If the conditions are true, it means that the program has accepted the syntax analysis. If the conditions are false, it means that the program has rejected the syntax analysis and the equation is syntactically wrong. The program will also end.



```python
def maincheck(self):
    for x in range(len(self.text)):
        # Reset variables
        self.a = ""
        self.stackString = ""

        # Print action
        print(self.action)

        # Pushing into stack
        self.stack.append(self.text[x][0])
        # Make all the elements in the stack a string
        # so that it is easier to print
        for i in range(len(self.stack)):
            self.stackString = self.stackString + self.stack[i]

        # Move forward the pointer for the input string
        self.inputElement = self.inputElement + 1
        # Make all the elements in the the input array a
        # string so that it is easier to print
        for i in range(len(self.text) - self.inputElement):
            self.a = self.a + self.text[i + self.inputElement][0]

        # Print stack and input
        print("$" + self.stackString + "\t" + self.a + "$" + "\t", end='')

        self.checkrules()
```

Figure 16: SA Method 8

The eighth method is called *maincheck*. This method takes in only one parameter which is *self*. This method runs a for-loop which removes items one-by-one from the tokens and adds it to the stack. After each removal, *self.checkrules* method is called to see if any production rules exists in the top of the stack.



```python
def parser(self):
    for x in range(len(self.text) - self.inputElement):
        self.a = self.a + self.text[x + self.inputElement][0]
    print("stack \t input \t action")
    print("$ \t" + self.a + "$" + "\t", end='')

    # Main function for shift reduce parser
    self.maincheck()

    # Check for production rules one last time
    self.checkrules()

    # Check if syntax is correct or not
    self.checkvalid()
```

Figure 17: SA Method 9

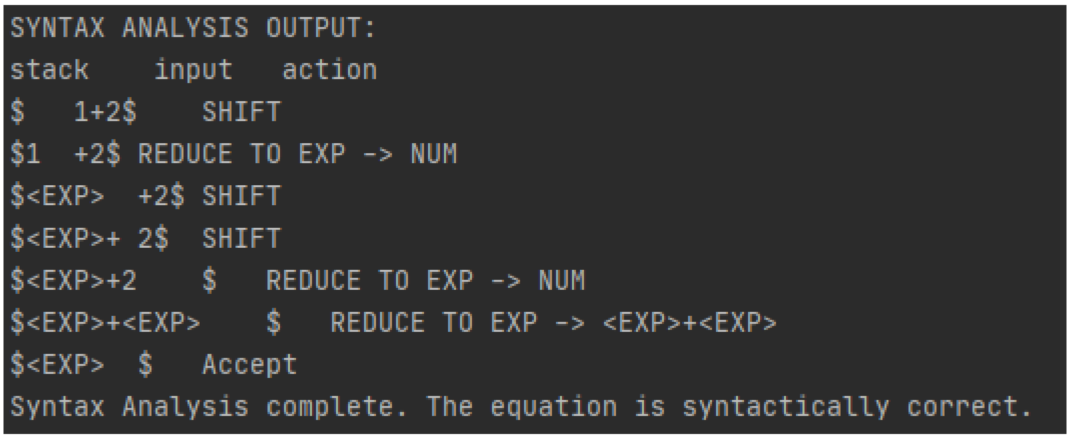
The ninth method, which is also the final one, is called *parser*. This method simply prints the initial words “stack”, “input” and “action”. After that *self.maincheck*, *self.checkrules* and *self.checkvalid* methods are called, which have already been explained above. This method simply is the one that starts the syntax analysis program.

Examples of syntax analysis for syntactically correct equations:

Example 1

Equation: 1+2

Output:



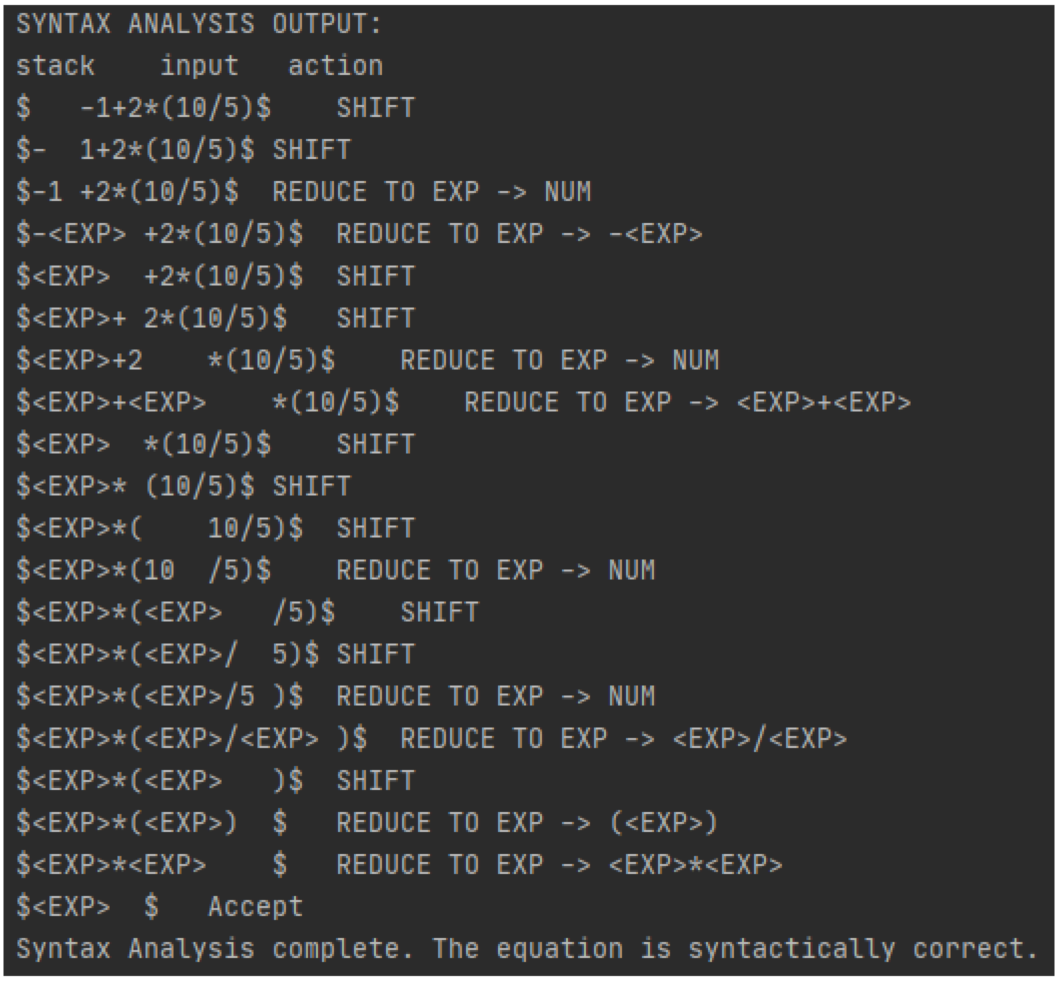
```
SYNTAX ANALYSIS OUTPUT:
stack    input   action
$   1+2$    SHIFT
$1  +2$ REDUCE TO EXP -> NUM
$<EXP>  +2$ SHIFT
$<EXP>+ 2$  SHIFT
$<EXP>+2    $   REDUCE TO EXP -> NUM
$<EXP>+<EXP>    $   REDUCE TO EXP -> <EXP>+<EXP>
$<EXP>  $   Accept
Syntax Analysis complete. The equation is syntactically correct.
```

Figure 18: SA Example 1

Example 2

Equation: -1+2*(10/5)

Output:



```
SYNTAX ANALYSIS OUTPUT:
stack    input   action
$   -1+2*(10/5)$    SHIFT
$-  1+2*(10/5)$ SHIFT
$-1 +2*(10/5)$  REDUCE TO EXP -> NUM
$-<EXP> +2*(10/5)$  REDUCE TO EXP -> -<EXP>
$<EXP>  +2*(10/5)$  SHIFT
$<EXP>+ 2*(10/5)$   SHIFT
$<EXP>+2    *(10/5)$    REDUCE TO EXP -> NUM
$<EXP>+<EXP>    *(10/5)$    REDUCE TO EXP -> <EXP>+<EXP>
$<EXP>  *(10/5)$    SHIFT
$<EXP>* (10/5)$ SHIFT
$<EXP>*(    10/5)$  SHIFT
$<EXP>*(10  /5)$    REDUCE TO EXP -> NUM
$<EXP>*(<EXP>   /5)$    SHIFT
$<EXP>*(<EXP>/  5)$ SHIFT
$<EXP>*(<EXP>/5 )$  REDUCE TO EXP -> NUM
$<EXP>*(<EXP>/<EXP> )$  REDUCE TO EXP -> <EXP>/<EXP>
$<EXP>*(<EXP>   )$  SHIFT
$<EXP>*(<EXP>)  $   REDUCE TO EXP -> (<EXP>)
$<EXP>*<EXP>    $   REDUCE TO EXP -> <EXP>*<EXP>
$<EXP>  $   Accept
Syntax Analysis complete. The equation is syntactically correct.
```

Figure 19: SA Example 2

Examples of syntax analysis for syntactically incorrect equations:

Example 1

Equation: 1+2+

Output:

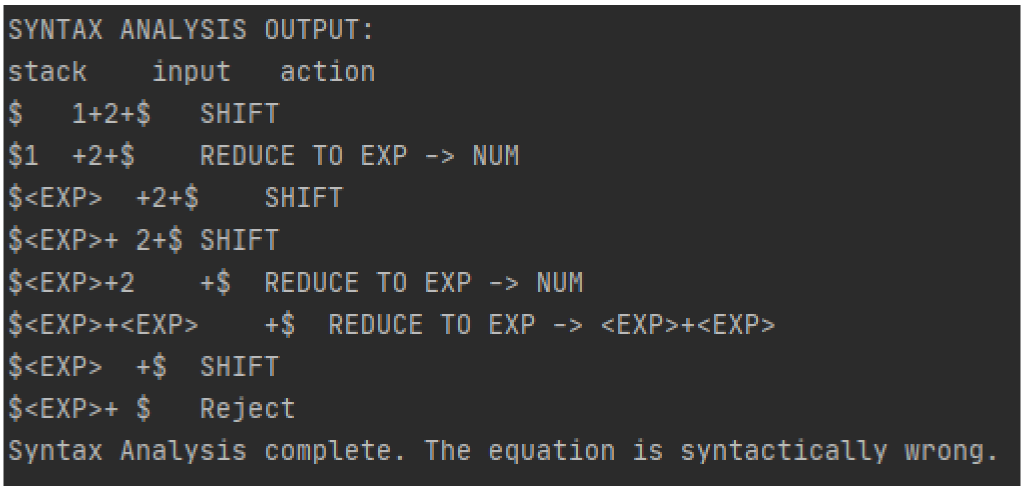

```
SYNTAX ANALYSIS OUTPUT:
stack    input   action
$   1+2+$   SHIFT
$1  +2+$    REDUCE TO EXP -> NUM
$<EXP>  +2+$    SHIFT
$<EXP>+ 2+$ SHIFT
$<EXP>+2    +$  REDUCE TO EXP -> NUM
$<EXP>+<EXP>    +$  REDUCE TO EXP -> <EXP>+<EXP>
$<EXP>  +$  SHIFT
$<EXP>+ $   Reject
Syntax Analysis complete. The equation is syntactically wrong.
```

Figure 20: SA Example 3

Example 2

Equation: 3*4+(-)

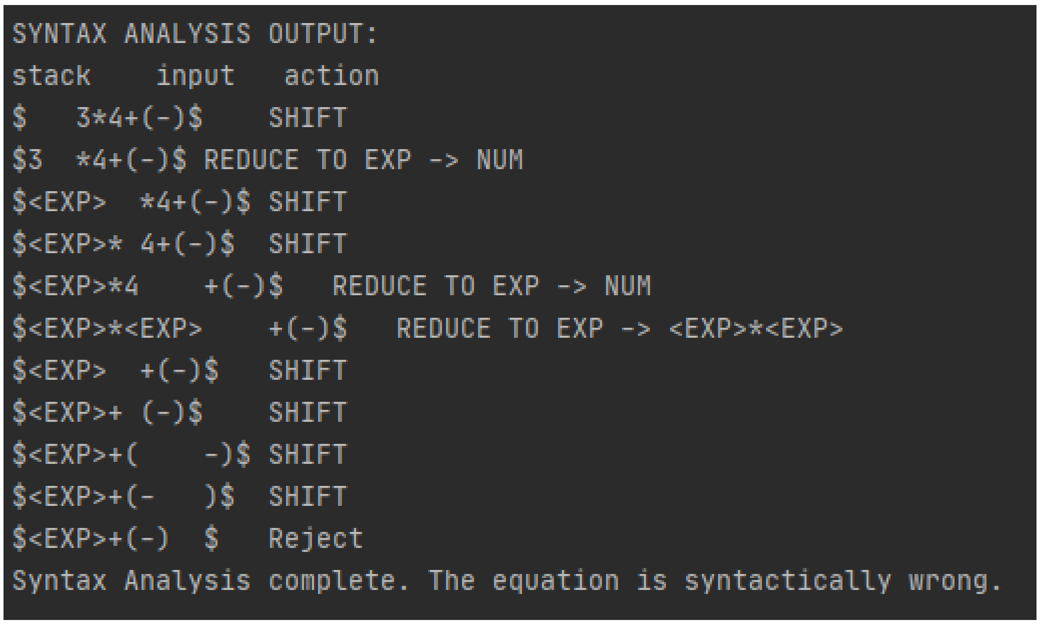
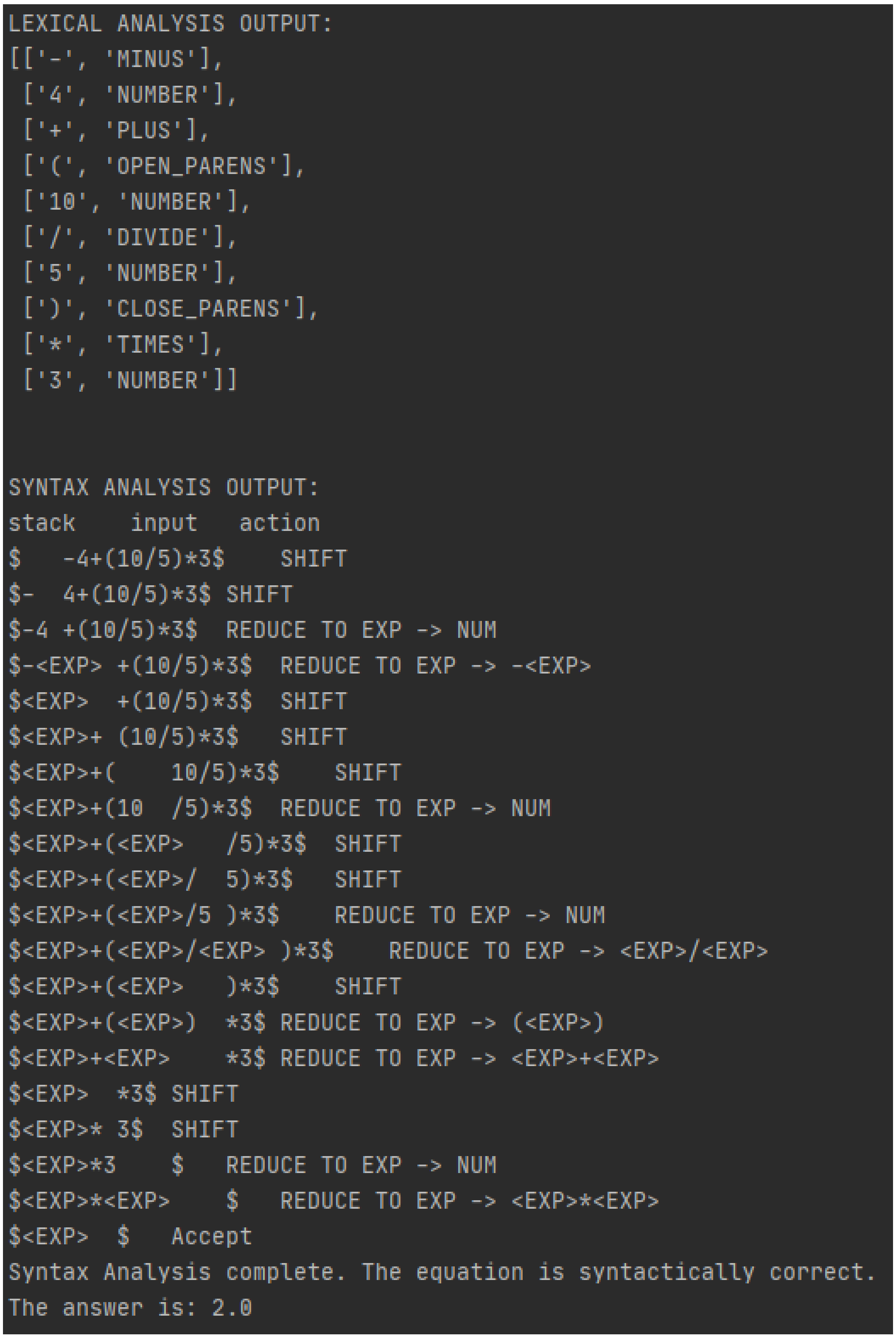

```
SYNTAX ANALYSIS OUTPUT:
stack    input   action
$   3*4+(-)$    SHIFT
$3  *4+(-)$ REDUCE TO EXP -> NUM
$<EXP>  *4+(-)$ SHIFT
$<EXP>* 4+(-)$  SHIFT
$<EXP>*4    +(-)$   REDUCE TO EXP -> NUM
$<EXP>*<EXP>    +(-)$   REDUCE TO EXP -> <EXP>*<EXP>
$<EXP>  +(-)$   SHIFT
$<EXP>+ (-)$    SHIFT
$<EXP>+(    -)$ SHIFT
$<EXP>+(-   )$  SHIFT
$<EXP>+(-)  $   Reject
Syntax Analysis complete. The equation is syntactically wrong.
```

Figure 21: SA Example 4

The program also calculates the output of the equation. The full program is shown in the figure below.

Equation: -4+(10/5)*3



```
LEXICAL ANALYSIS OUTPUT:
[['-', 'MINUS'],
 ['4', 'NUMBER'],
 ['+', 'PLUS'],
 ['(', 'OPEN_PARENS'],
 ['10', 'NUMBER'],
 ['/', 'DIVIDE'],
 ['5', 'NUMBER'],
 [')', 'CLOSE_PARENS'],
 ['*', 'TIMES'],
 ['3', 'NUMBER']]



SYNTAX ANALYSIS OUTPUT:
stack    input   action
$   -4+(10/5)*3$    SHIFT
$-  4+(10/5)*3$ SHIFT
$-4 +(10/5)*3$  REDUCE TO EXP -> NUM
$-<EXP> +(10/5)*3$  REDUCE TO EXP -> -<EXP>
$<EXP>  +(10/5)*3$  SHIFT
$<EXP>+ (10/5)*3$   SHIFT
$<EXP>+(    10/5)*3$    SHIFT
$<EXP>+(10  /5)*3$  REDUCE TO EXP -> NUM
$<EXP>+(<EXP>   /5)*3$  SHIFT
$<EXP>+(<EXP>/  5)*3$   SHIFT
$<EXP>+(<EXP>/5 )*3$    REDUCE TO EXP -> NUM
$<EXP>+(<EXP>/<EXP> )*3$    REDUCE TO EXP -> <EXP>/<EXP>
$<EXP>+(<EXP>   )*3$    SHIFT
$<EXP>+(<EXP>)  *3$ REDUCE TO EXP -> (<EXP>)
$<EXP>+<EXP>    *3$ REDUCE TO EXP -> <EXP>+<EXP>
$<EXP>  *3$ SHIFT
$<EXP>* 3$  SHIFT
$<EXP>*3    $   REDUCE TO EXP -> NUM
$<EXP>*<EXP>    $   REDUCE TO EXP -> <EXP>*<EXP>
$<EXP>  $   Accept
Syntax Analysis complete. The equation is syntactically correct.
The answer is: 2.0
```

Figure 22: Complete Program Output