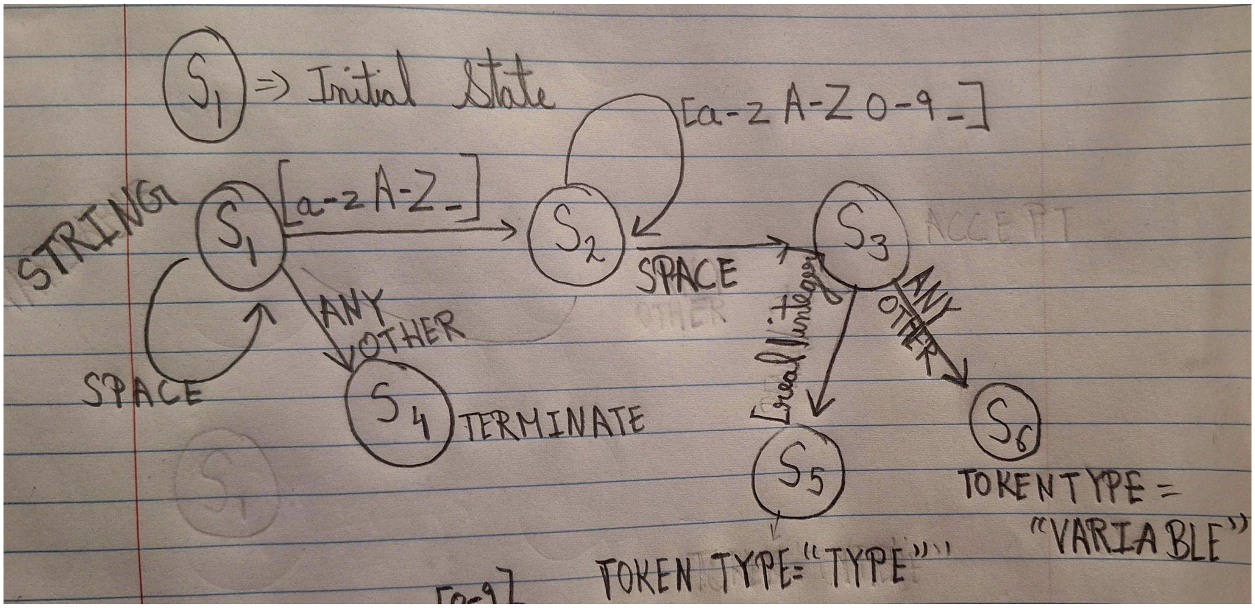
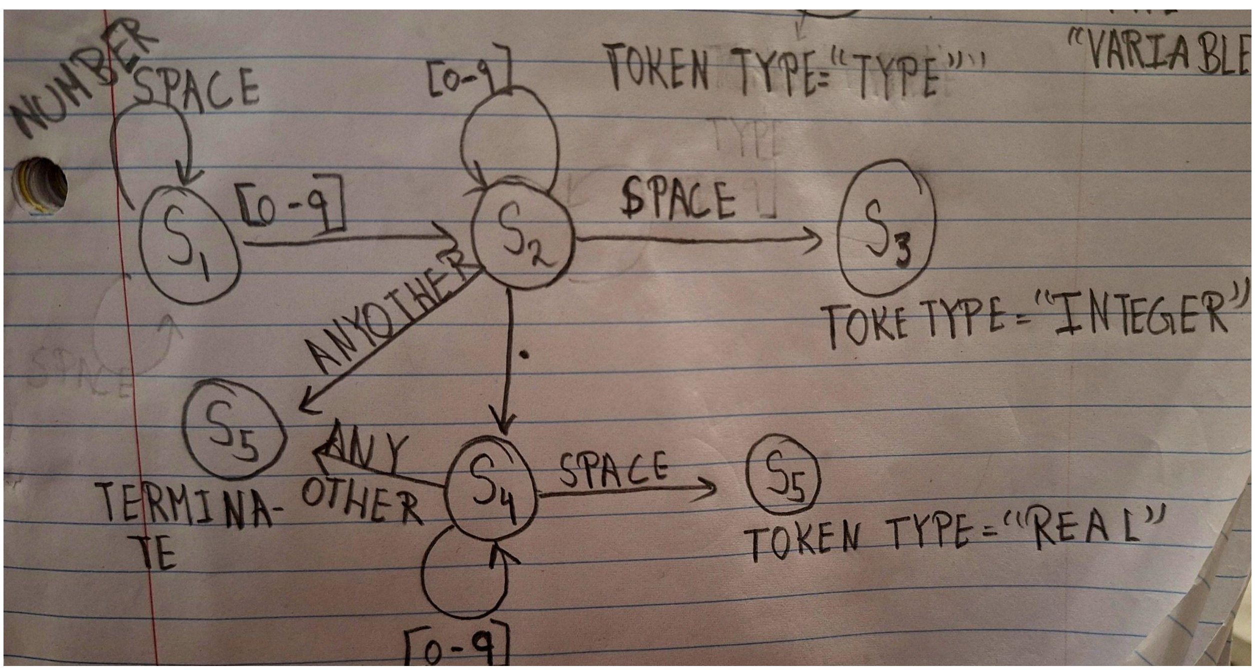
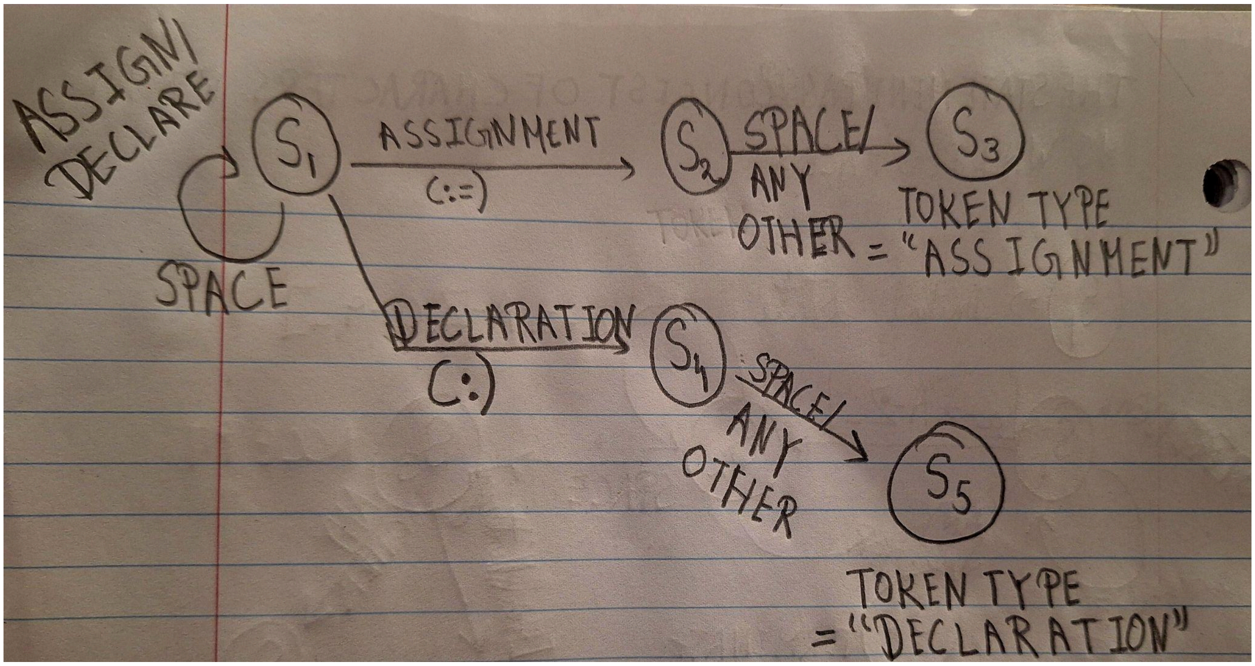
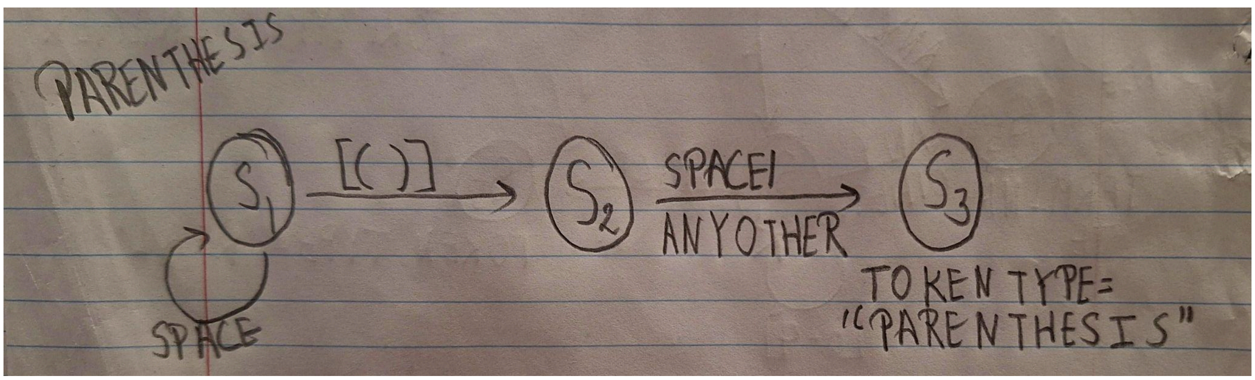
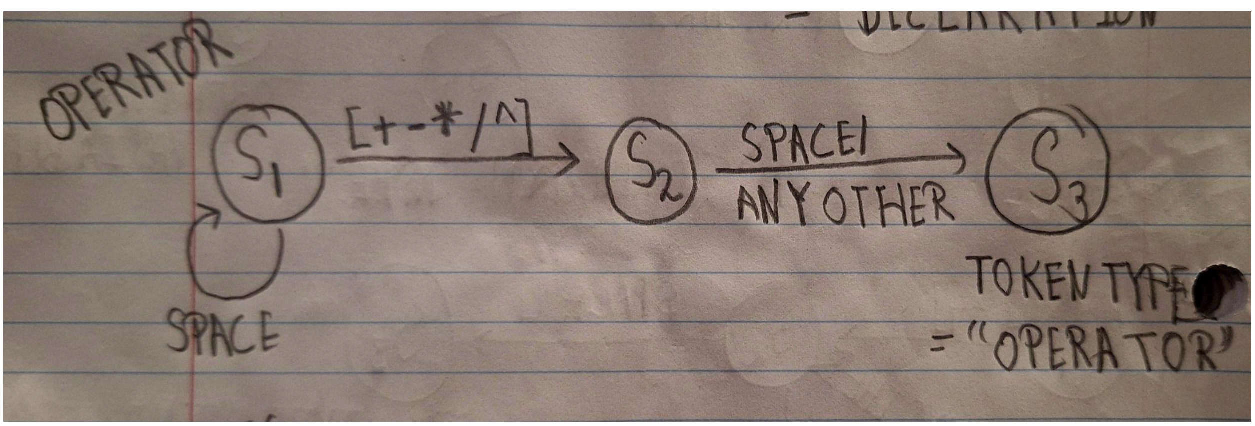
Lexical Analyzer

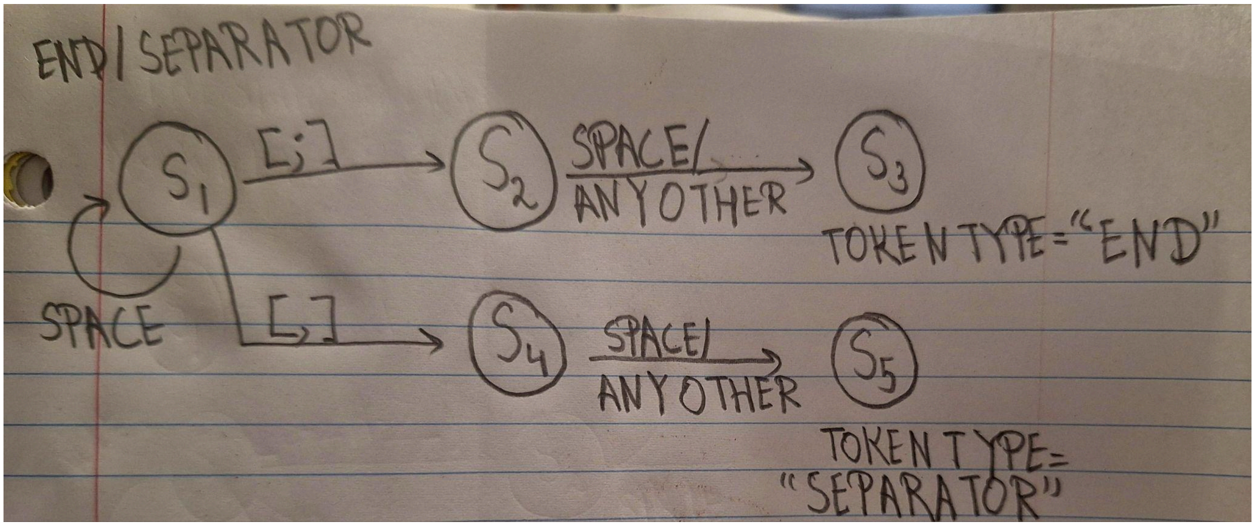





ASSIGN/ DECLARE
S1
ASSIGNMENT (:=)
S2
SPACE/ ANY OTHER
S3
TOKEN TYPE = "ASSIGNMENT"
SPACE
DECLARATION (:)
S4
SPACE/ ANY OTHER
S5
TOKEN TYPE = "DECLARATION"


PARENTHESIS
S1
[( )]
S2
SPACE/ ANYOTHER
S3
TOKEN TYPE= "PARENTHESIS"
SPACE


OPERATOR
S1
[+-*/^]
S2
SPACE/ ANYOTHER
S3
TOKEN TYPE = "OPERATOR"
SPACE



Code:

```
def lexical_analyzer(input_string):
    """
    Tokenizes input string into supported tokens and validates them.
    """
    # Using regular expressions to differentiate the tokens
    tokens = []
    patterns = {
        "VARIABLE": r"[a-zA-Z_][a-zA-Z0-9_]*",
        "INTEGER": r"\d+",
        "REAL": r"\d+\.\d+",
        "TYPE": r"(integer|real)",
        "ASSIGNMENT": r":=",
        "OPERATOR": r"[+\-*^/]",
        "END": r"\;",
        "DECLARATION": r"\:",
        "OPEN PARENTHESIS": r"\(",
        "CLOSE PARENTHESIS": r"\)",
        "SEPARATOR": r","
    }

    for token in
re.findall(r"[a-zA-Z_][a-zA-Z0-9_]*|\d+\.\d+|\d+|:=|[+\-*/^]|;|:|\(|\)|,",
input_string):
        if re.fullmatch(patterns["TYPE"], token):
            tokens.append(("TYPE", token))
        elif re.fullmatch(patterns["VARIABLE"], token):
            tokens.append(("VARIABLE", token))
        elif re.fullmatch(patterns["INTEGER"], token):
            tokens.append(("INTEGER", token))
        elif re.fullmatch(patterns["REAL"], token):
```

```
            tokens.append(("REAL", token))
        elif re.fullmatch(patterns["ASSIGNMENT"], token):
            tokens.append(("ASSIGNMENT", token))
        elif re.fullmatch(patterns["OPERATOR"], token):
            tokens.append(("OPERATOR", token))
        elif re.fullmatch(patterns["END"], token):
            tokens.append(("END", token))
        elif re.fullmatch(patterns["DECLARATION"], token):
            tokens.append(("DECLARATION", token))
        elif re.fullmatch(patterns["OPEN PARENTHESIS"], token):
            tokens.append(("OPEN PARENTHESIS", token))
        elif re.fullmatch(patterns["CLOSE PARENTHESIS"], token):
            tokens.append(("CLOSE PARENTHESIS", token))
        elif re.fullmatch(patterns["SEPARATOR"], token):
            tokens.append(("SEPARATOR", token))
        else:
            raise ValueError(f"Invalid token: {token}")

    return tokens
```

Syntax Analyzer

Grammar used for Syntax Analyzer:

```
Numeric -> [0-9]*
Operator -> *|+|-|/|^
Variable -> [a-zA-Z_][A-Za-z_0-9]*
Type -> integer | real
Statement_list -> Statement*
Statement -> Declaration | Assignment
Declaration -> Variable* : Type ;
Assignment -> Variable := Expression ;
Expression -> Addition
Addition -> Multiply(+|-)Multiply
Multiply -> Exponent(*|/)Exponent
Exponent-> Base(^)Base
Base -> Expression | lambda (INTEGER | REAL | VARIABLE | OPEN PARENTHESIS |
CLOSE PARENTHESIS)
```

In the Below code parse_statement handles both the cases for assignment as well as declaration in the programming language. Afterwards parse_addition, parse_multiplication, parse_exponent and parse_base are called in order to make a tree like structure to verify the structure of the inputted code.

```
class SyntaxAnalyzer:
   def __init__(self, tokens):
       self.tokens = tokens
       self.current_token_index = 0
       self.symbol_table = {}  # To store declared variables and their types

   def get_current_token(self):
       '''Obtaining the current token and incrementing the index'''
       if self.current_token_index < len(self.tokens):
           token = self.tokens[self.current_token_index]
           self.current_token_index += 1
           return token
       return None

   def parse(self):
       """
       Grammar:
           Numeric -> [0-9]*
           Operator -> *|+|-|/|^
           Variable -> [a-zA-Z_][A-Za-z_0-9]*
           Type -> integer | real
           Statement_list -> Statement*
           Statement -> Declaration | Assignment
```

```
            Declaration -> Variable* : Type ;
            Assignment -> Variable := Expression ;
            Expression -> Addition
            Addition -> Multiply(+|-)Multiply
            Multiply -> Exponent(*|/)Exponent
            Exponent-> Base(^)Base
            Base -> Expression | lambda (INTEGER | REAL | VARIABLE | OPEN
PARENTHESIS | CLOSE PARENTHESIS)
        """
        self.parse_statement_list()

    def parse_statement_list(self):
        '''Process until no tokens are left'''
        while self.current_token_index < len(self.tokens):
            self.parse_statement()

    def parse_statement(self):
        '''Check syntax validity and handle parsing expressions'''
        token = self.get_current_token()
        if token is None:
            return
        token_type, token_value = token

        if token_type == "VARIABLE":
            saved_var = token_value  # Save the variable name
            token_type, token_value = self.get_current_token()

            # Declaration
            if token_type == "SEPARATOR":
                variable_list = [saved_var]
                while True:
                    token_type, token_value = self.get_current_token()

                    if token_type == "VARIABLE":
                        variable_list.append(token_value)
                    elif token_type == "DECLARATION":
                        break
                    elif token_type != "SEPARATOR":
                        raise SyntaxError(f"Expected SEPARATOR, got:
{token_type}")

                token_type, token_value = self.get_current_token()
                if token_type == "TYPE":
                    for var in variable_list:
                        self.symbol_table[var] = token_value
                    token_type, token_value = self.get_current_token()
                    if token_type != "END":
                        raise SyntaxError(f"Expected END, got: {token_type}")
                else:
```

```
                raise SyntaxError(f"Expected TYPE, got: {token_type}")

        # Assignment
        elif token_type == "ASSIGNMENT":
            self.parse_addition()
            token_type, token_value = self.get_current_token()
            if token_type != "END":
                raise SyntaxError(f"Expected END, got: {token_type}")
        else:
            raise SyntaxError(f"Unexpected token: {token_value}")
    else:
        raise SyntaxError(f"Unexpected token: {token_value}")

def parse_addition(self):
    self.parse_multiplication()
    while self.current_token_index < len(self.tokens):
        type_token, token = self.tokens[self.current_token_index]
        if type_token == "OPERATOR" and token in "+-":
            self.get_current_token()
            self.parse_multiplication()
        else:
            break

def parse_multiplication(self):
    self.parse_exponent()
    while self.current_token_index < len(self.tokens):
        type_token, token = self.tokens[self.current_token_index]
        if type_token == "OPERATOR" and token in "*/":
            self.get_current_token()
            self.parse_exponent()
        else:
            break

def parse_exponent(self):
    self.parse_base()
    while self.current_token_index < len(self.tokens):
        type_token, token = self.tokens[self.current_token_index]
        if type_token == "OPERATOR" and token == "^":
            self.get_current_token()
            self.parse_base()
        else:
            break

def parse_base(self):
    if self.current_token_index < len(self.tokens):
        type_token, token = self.get_current_token()

        if type_token == "OPEN_PARENTHESIS":
            self.parse_addition()
```

```
                type_token, token = self.get_current_token()
                if type_token != "CLOSE PARENTHESIS":
                    raise SyntaxError("Expected CLOSE PARENTHESIS")
            elif type_token not in ("VARIABLE", "REAL", "INTEGER"):
                raise SyntaxError(f"Expected VARIABLE | REAL | INTEGER, found
{type_token}")
```

Complete Code Implementation:

```
import re

# Lexical Analyzer
def lexical_analyzer(input_string):
    """
    Tokenizes input string into supported tokens and validates them.
    """
    # Using regular expressions to differentiate the tokens
    tokens = []
    patterns = {
        "VARIABLE": r"[a-zA-Z_][a-zA-Z0-9_]*",
        "INTEGER": r"\d+",
        "REAL": r"\d+\.\d+",
        "TYPE": r"(integer|real)",
        "ASSIGNMENT": r":=",
        "OPERATOR": r"[+\-*^/]",
        "END": r"\;",
        "DECLARATION": r"\:",
        "OPEN PARENTHESIS": r"\(",
        "CLOSE PARENTHESIS": r"\)",
        "SEPARATOR": r","
    }

    for token in
re.findall(r"[a-zA-Z_][a-zA-Z0-9_]*|\d+\.\d+|\d+|:=|[+\-*/^]|;|:|\(|\)|,",
input_string):
        if re.fullmatch(patterns["TYPE"], token):
            tokens.append(("TYPE", token))
        elif re.fullmatch(patterns["VARIABLE"], token):
            tokens.append(("VARIABLE", token))
        elif re.fullmatch(patterns["INTEGER"], token):
            tokens.append(("INTEGER", token))
        elif re.fullmatch(patterns["REAL"], token):
            tokens.append(("REAL", token))
        elif re.fullmatch(patterns["ASSIGNMENT"], token):
            tokens.append(("ASSIGNMENT", token))
        elif re.fullmatch(patterns["OPERATOR"], token):
            tokens.append(("OPERATOR", token))
        elif re.fullmatch(patterns["END"], token):
            tokens.append(("END", token))
        elif re.fullmatch(patterns["DECLARATION"], token):
            tokens.append(("DECLARATION", token))
        elif re.fullmatch(patterns["OPEN PARENTHESIS"], token):
            tokens.append(("OPEN PARENTHESIS", token))
        elif re.fullmatch(patterns["CLOSE PARENTHESIS"], token):
            tokens.append(("CLOSE PARENTHESIS", token))
        elif re.fullmatch(patterns["SEPARATOR"], token):
            tokens.append(("SEPARATOR", token))
```

```python
        else:
            raise ValueError(f"Invalid token: {token}")

    return tokens


class SyntaxAnalyzer:
    def __init__(self, tokens):
        self.tokens = tokens
        self.current_token_index = 0
        self.symbol_table = {}  # To store declared variables and their types

    def get_current_token(self):
        '''Obtaining the current token and incrementing the index'''
        if self.current_token_index < len(self.tokens):
            token = self.tokens[self.current_token_index]
            self.current_token_index += 1
            return token
        return None

    def parse(self):
        """
        Grammar:
            Numeric -> [0-9]*
            Operator -> *|+|-|/|^
            Variable -> [a-zA-Z_][A-Za-z_0-9]*
            Type -> integer | real
            Statement_list -> Statement*
            Statement -> Declaration | Assignment
            Declaration -> Variable* : Type ;
            Assignment -> Variable := Expression ;
            Expression -> Addition
            Addition -> Multiply(+|-)Multiply
            Multiply -> Exponent(*|/)Exponent
            Exponent-> Base(^)Base
            Base -> Expression | lambda (INTEGER | REAL | VARIABLE | OPEN
PARENTHESIS | CLOSE PARENTHESIS)
        """
        self.parse_statement_list()

    def parse_statement_list(self):
        '''Process until no tokens are left'''
        while self.current_token_index < len(self.tokens):
            self.parse_statement()

    def parse_statement(self):
        '''Check syntax validity and handle parsing expressions'''
        token = self.get_current_token()
        if token is None:
```

```
            return
        token_type, token_value = token

        if token_type == "VARIABLE":
            saved_var = token_value  # Save the variable name
            token_type, token_value = self.get_current_token()

            # Declaration
            if token_type == "SEPARATOR":
                variable_list = [saved_var]
                while True:
                    token_type, token_value = self.get_current_token()

                    if token_type == "VARIABLE":
                        variable_list.append(token_value)
                    elif token_type == "DECLARATION":
                        break
                    elif token_type != "SEPARATOR":
                        raise SyntaxError(f"Expected SEPARATOR, got:
{token_type}")

                token_type, token_value = self.get_current_token()
                if token_type == "TYPE":
                    for var in variable_list:
                        self.symbol_table[var] = token_value
                    token_type, token_value = self.get_current_token()
                    if token_type != "END":
                        raise SyntaxError(f"Expected END, got: {token_type}")
                else:
                    raise SyntaxError(f"Expected TYPE, got: {token_type}")

            # Assignment
            elif token_type == "ASSIGNMENT":
                self.parse_addition()
                token_type, token_value = self.get_current_token()
                if token_type != "END":
                    raise SyntaxError(f"Expected END, got: {token_type}")
            else:
                raise SyntaxError(f"Unexpected token: {token_value}")
        else:
            raise SyntaxError(f"Unexpected token: {token_value}")

    def parse_addition(self):
        self.parse_multiplication()
        while self.current_token_index < len(self.tokens):
            type_token, token = self.tokens[self.current_token_index]
            if type_token == "OPERATOR" and token in "+-":
                self.get_current_token()
                self.parse_multiplication()
```

```
            else:
                break

    def parse_multiplication(self):
        self.parse_exponent()
        while self.current_token_index < len(self.tokens):
            type_token, token = self.tokens[self.current_token_index]
            if type_token == "OPERATOR" and token in "*/":
                self.get_current_token()
                self.parse_exponent()
            else:
                break

    def parse_exponent(self):
        self.parse_base()
        while self.current_token_index < len(self.tokens):
            type_token, token = self.tokens[self.current_token_index]
            if type_token == "OPERATOR" and token == "^":
                self.get_current_token()
                self.parse_base()
            else:
                break

    def parse_base(self):
        if self.current_token_index < len(self.tokens):
            type_token, token = self.get_current_token()

            if type_token == "OPEN PARENTHESIS":
                self.parse_addition()
                type_token, token = self.get_current_token()
                if type_token != "CLOSE PARENTHESIS":
                    raise SyntaxError("Expected CLOSE PARENTHESIS")
            elif type_token not in ("VARIABLE", "REAL", "INTEGER"):
                raise SyntaxError(f"Expected VARIABLE | REAL | INTEGER, found 
{type_token}")



# Main program
if __name__ == "__main__":
    # Input program
    program = """
    apple, banana, papaya : integer;
    a := a - (4 ^ 4);
    b := papaya + (apple - banana^5);
    i, j, k: real;
    i := 123;
    a := b;
```

```python
"""
print("Input Program:")
print(program)

# Lexical Analysis
print("\nLexical Analysis:")
try:
    tokens = lexical_analyzer(program)
    for token in tokens:
        print(token)
except ValueError as e:
    print(f"Lexical Error: {e}")

# Syntax Analysis
print("\nSyntax Analysis:")
try:
    parser = SyntaxAnalyzer(tokens)
    parser.parse()
    print("Syntax Analysis Completed Successfully!")
    print("Symbol Table:", parser.symbol_table)
except SyntaxError as e:
    print(f"Syntax Error: {e}")
```

Correct Output:
Input Program:

apple, banana, papaya : integer;
a := a - (4 ^ 4);
b := papaya + (apple - banana^5);
i, j, k: real;
i := 123;
a := b;

Lexical Analysis:
('VARIABLE', 'apple')
('SEPARATOR', ',')
('VARIABLE', 'banana')
('SEPARATOR', ',')
('VARIABLE', 'papaya')
('DECLARATION', ':')
('TYPE', 'integer')
('END', ';')
('VARIABLE', 'a')
('ASSIGNMENT', ':=')

```
('VARIABLE', 'a')
('OPERATOR', '-')
('OPEN PARENTHESIS', '(')
('INTEGER', '4')
('OPERATOR', '^')
('INTEGER', '4')
('CLOSE PARENTHESIS', ')')
('END', ';')
('VARIABLE', 'b')
('ASSIGNMENT', ':=')
('VARIABLE', 'papaya')
('OPERATOR', '+')
('OPEN PARENTHESIS', '(')
('VARIABLE', 'apple')
('OPERATOR', '-')
('VARIABLE', 'banana')
('OPERATOR', '^')
('INTEGER', '5')
('CLOSE PARENTHESIS', ')')
('END', ';')
('VARIABLE', 'i')
('SEPARATOR', ',')
('VARIABLE', 'j')
('SEPARATOR', ',')
('VARIABLE', 'k')
('DECLARATION', ':')
('TYPE', 'real')
('END', ';')
('VARIABLE', 'i')
('ASSIGNMENT', ':=')
('INTEGER', '123')
('END', ';')
('VARIABLE', 'a')
('ASSIGNMENT', ':=')
('VARIABLE', 'b')
('END', ';')

Syntax Analysis:
Syntax Analysis Completed Successfully!
Symbol Table: {'apple': 'integer', 'banana': 'integer', 'papaya': 'integer', 'i': 'real', 'j': 'real', 'k': 'real'}

Process finished with exit code 0
```

Incorrect Output:

First:
Input Program:

```
a := a - 4 ^ 4);
```

Lexical Analysis:
('VARIABLE', 'a')
('ASSIGNMENT', ':=')
('VARIABLE', 'a')
('OPERATOR', '-')
('INTEGER', '4')
('OPERATOR', '^')
('INTEGER', '4')
('CLOSE PARENTHESIS', ')')
('END', ';')

Syntax Analysis:
Syntax Error: Expected END, got: CLOSE PARENTHESIS

Second:
Input Program:

```
a := a -  ^ 4);
```

Lexical Analysis:
('VARIABLE', 'a')
('ASSIGNMENT', ':=')
('VARIABLE', 'a')
('OPERATOR', '-')
('OPERATOR', '^')
('INTEGER', '4')
('CLOSE PARENTHESIS', ')')
('END', ';')

Syntax Analysis:
Syntax Error: Expected VARIABLE | REAL | INTEGER, found OPERATOR

Process finished with exit code 0

Third:

Input Program:

```
a, b : rabbit;
```

Lexical Analysis:
('VARIABLE', 'a')
('SEPARATOR', ',')
('VARIABLE', 'b')
('DECLARATION', ':')
('VARIABLE', 'rabbit')
('END', ';')

Syntax Analysis:
Syntax Error: Expected TYPE, got: VARIABLE

Process finished with exit code 0

Fourth:

Input Program:

```
a*b := 10;
```

Lexical Analysis:
('VARIABLE', 'a')
('OPERATOR', '*')
('VARIABLE', 'b')
('ASSIGNMENT', ':=')
('INTEGER', '10')
('END', ';')

Syntax Analysis:
Syntax Error: Unexpected token: *

Process finished with exit code 0

Fifth:

Input Program:

```
apple:= (9 * 10) + 8 SidedDice;
```

Lexical Analysis:
('VARIABLE', 'apple')
('ASSIGNMENT', ':=')
('OPEN PARENTHESIS', '(')
('INTEGER', '9')
('OPERATOR', '*')
('INTEGER', '10')
('CLOSE PARENTHESIS', ')')
('OPERATOR', '+')
('INTEGER', '8')
('VARIABLE', 'SidedDice')
('END', ';')

Syntax Analysis:
Syntax Error: Expected END, got: VARIABLE