Programming Language

Fall 2020

CSE302L–Systems Programming Lab

PROJECT REPORT

Submitted by:

Mian Inshaullah Zia (18PWCSE1721, Sec: C)

Fakeha Saeed (18PWCSE1692, Sec: A)

“On my honor, as a student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work.”

Student Signature: ______________, ______________, ______________

Submitted to:

Engr. Madiha Sher

Department of Computer Systems Engineering

Contents

INTRODUCTION

We’re making an interpreter for our own programming language. The mechanism behind Programming languages and compilers can be vague for beginners because they’re often taken for granted. Therefore, to strengthen our fundamentals of programming, we’ll be writing a programming language to understand aspects of the computer stack, starting from the CPU and Memory through the operating system, built-in data structures, processes, threads, libraries, applications, and the stack.

- A programming language allows us to write efficient programs and develop solutions such as applications, games, and software.
- We use programming to automate, maintain, assemble, measure, and interpret the processing of the data and information.
- It acts as a mediator between machines and humans
- To pursue the goals of clear and concise communication with the processor we’ve created a programming language.

How it Works

To create our own programming language, we’ll first create a Lexer followed by a Parser and an Interpreter. Then we’ll add the necessary mechanisms and functions for Power Operators, Variables, Logical Operators, Arguments such as IF, FOR, WHILE, functions, strings, lists, and multi-line statements. We’ll also add special built-in functions such as return, continue, and break.

FEATURES

Our Programming Language offers the following features:

- Lexer
- Parser
- Interpreter

CLASS DIAGRAM

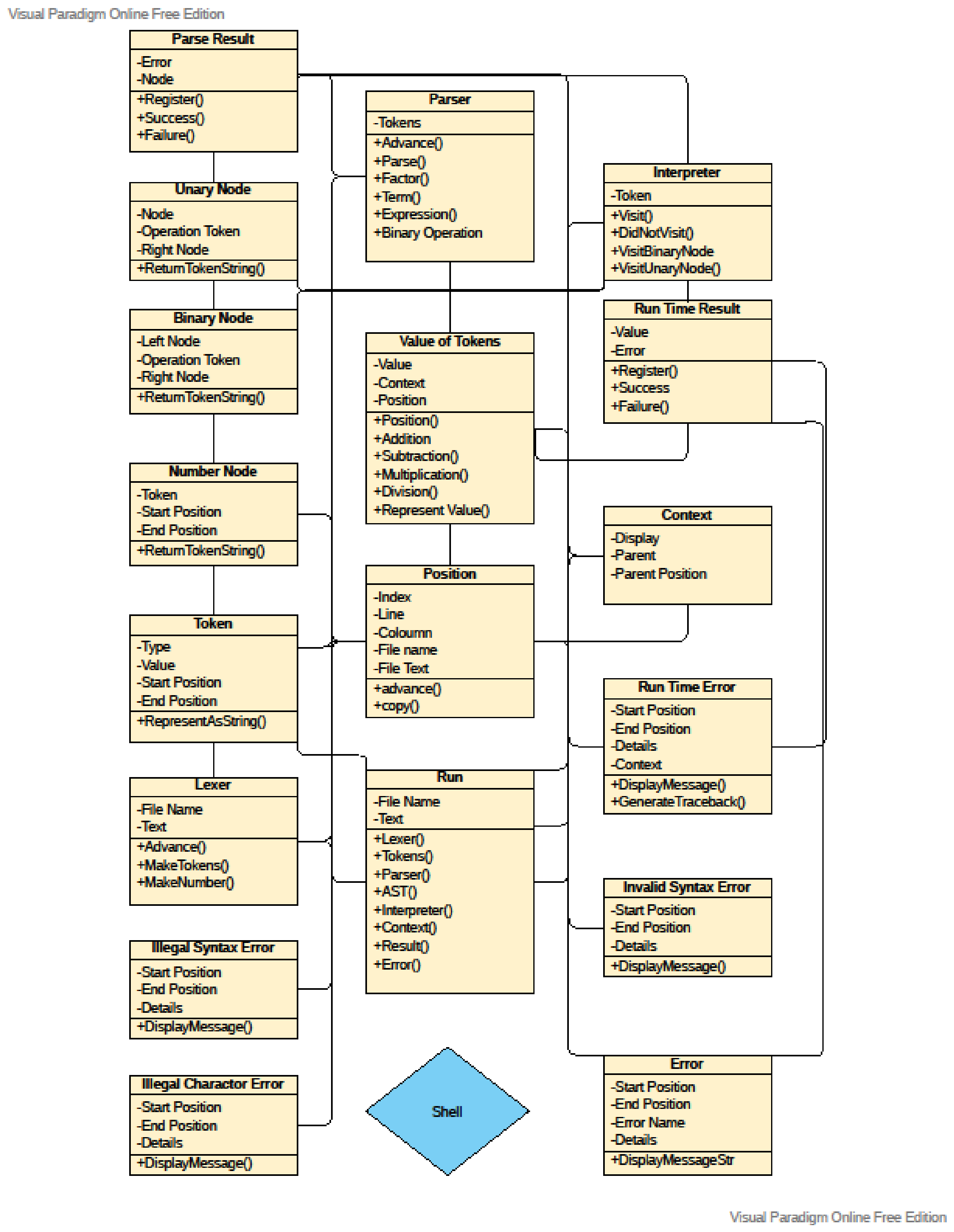


Figure 1.1 Class Diagram

Classes

There are a total of 16 classes in our code. We'll be discussing each class and its code. Our classes are divided into four subcategories.

1. Lexer
2. Parser
3. Interpreter
4. Error handling and Positional Analysis

Error Handling and Positional Analysis

To handle errors we've created a class that accurately tells us where the error is. At the moment, we've only defined three errors namely: IllegalCharError, InvalidSyntaxError, and RTError.

Moreover, we've also created a position class that monitors and returns the position of the error.

```
class Error:

    def __init__(self, pos_start, pos_end, error_name, details):
        self.pos_start = pos_start
        self.pos_end = pos_end
        self.error_name = error_name
        self.details = details

    def as_string(self):
        result = f'{self.error_name}: {self.details}\n'
        result += f'File {self.pos_start.fn}, line {self.pos_start.ln + 1}'
        #result += '\n\n' + string_with_arrows(self.pos_start.ftxt, self.pos_star
t, self.pos_end)
        return result
```
Figure 2.1 Error class

This is our main Error class. This error class essentially returns the result which returns the start position, the end position, the error name, and the details of the error in the code as a string.

To define different error types, we've created child classes that inherit the attributes and functions of the main Error class and returns the 'Error Type.'

```
class IllegalCharError(Error):
    def __init__(self, pos_start, pos_end, details):
        super().__init__(pos_start, pos_end, 'Illegal Character', details)
```
Figure 2.2 Illegal Character Error

Positional Analysis

```
class Position:
    def __init__(self, idx, ln, col, fn, ftxt):
        self.idx = idx
        self.ln = ln
        self.col = col
        self.fn = fn
        self.ftxt = ftxt

    def advance(self, current_char=None):
        self.idx += 1
        self.col += 1

        if current_char == '\n':
            self.ln += 1
            self.col = 0

        return self

    def copy(self):
        return Position(self.idx, self.ln, self.col, self.fn, self.ftxt)
```

Figure 2.3 Position Class

Here’s our Position class. It keeps track of the index, line, column, file name, and file text. It has two functions. The advance function increments the position based on the arguments it is presented with. The copy function, returns the index, line, column, file name, and file text.

Lexer

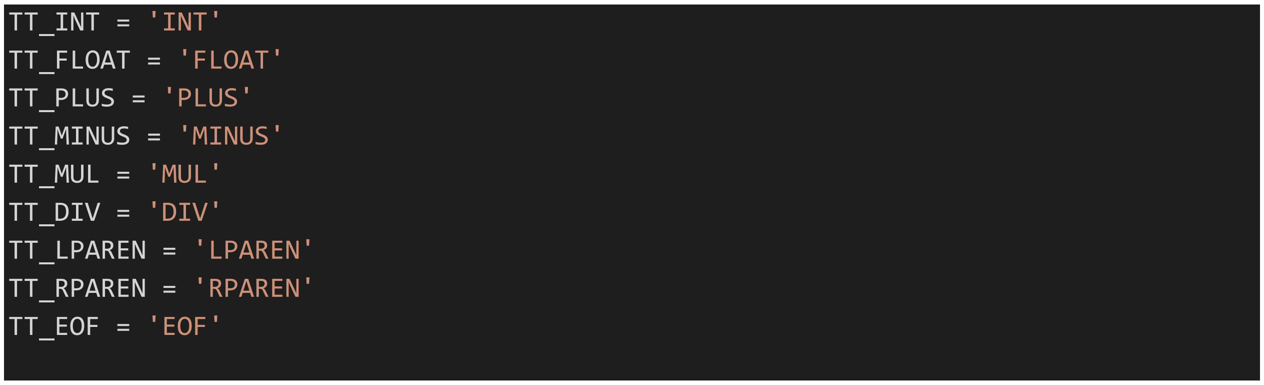
A lexical analyzer is an important component that takes a string and breaks it down into smaller units that can be understood by the Parser. It converts a ‘sequence’ of characters in a program to a ‘sequence’ of Tokens. Simply put, it serves as a scanner.

- For example: If we have pass 123 + 123 to a programming language. We pass it through as a string. The lexer will categorize each variable in the code to its corresponding tags hence the code will be tokenized as INT:123, PLUS, INT:123

How did we do it?

- We assigned numbers as constant so the Processor can recognize them as numbers.
- We created a Token class that works with the Position Class mentioned up.
- The Token class creates tokens specifying the category and the value.
- The Position class manages the position of the indexes in the Array.
- For example, when we pass 123 + 123. Our code accepts it as an array as [‘123’,’+’,’123]. To break down and tokenize each element of the array we need to track the position.
- After we created our Token class and Position Class
- We created a Lexer class that makes tokens, appends them into an array, and differentiates if the number is an Integer or Float.
- We specified number and the basic arithmetic operators to the Lexer so it can recognize it every time we pass it through.

Code



```
TT_INT = 'INT'
TT_FLOAT = 'FLOAT'
TT_PLUS = 'PLUS'
TT_MINUS = 'MINUS'
TT_MUL = 'MUL'
TT_DIV = 'DIV'
TT_LPAREN = 'LPAREN'
TT_RPAREN = 'RPAREN'
TT_EOF = 'EOF'
```

Figure 3.1 Defining Tokens

Here are our tokens. Before we start with the Lexer we need to define tokens for each instance. After we’ve done that, we can move forward and create a token class which will assign these tokens.

Class Token

```
class Token:
    def __init__(self, type_, value=None, pos_start=None, pos_end=None):
        self.type = type_
        self.value = value

        if pos_start:
            self.pos_start = pos_start.copy()
            self.pos_end = pos_start.copy()
            self.pos_end.advance()

        if pos_end:
            self.pos_end = pos_end

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

Figure 3.2 Token Class

The token class keeps track of the type, the value, and the position of the code. After it has tracked the position, it returns the type of the token as a string.

Class Lexer

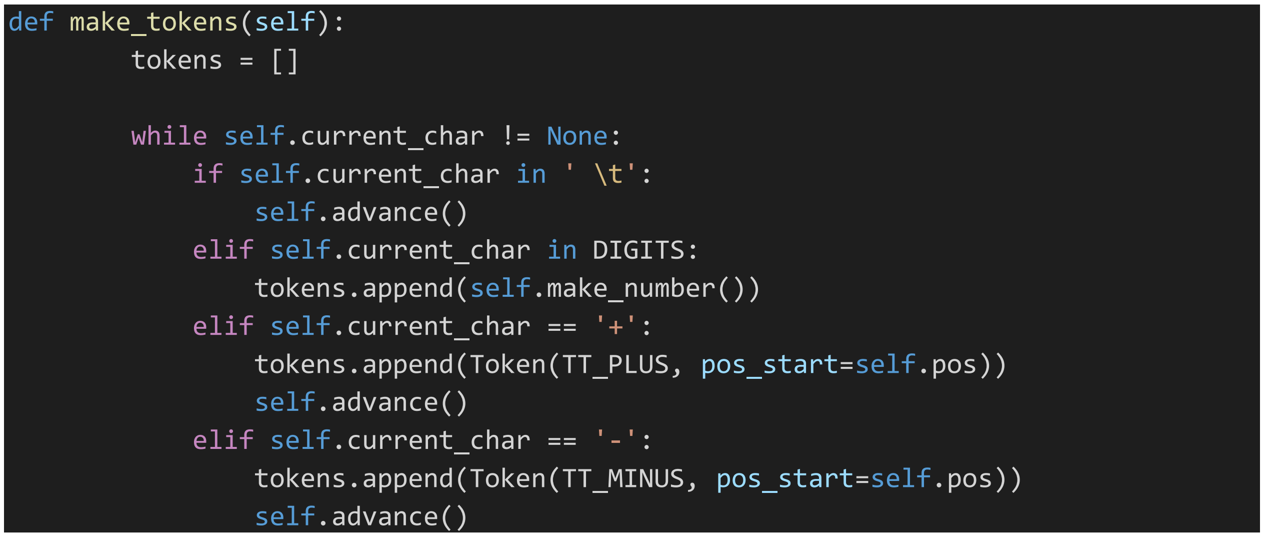
The lexer class essentially makes tokens, increments each part of the code, and returns the position. It has three functions. The advance function increments through the code. The make token function assigns tokens. And the Make number tokens sees if the number is an integer or a float.

```
class Lexer:
    def __init__(self, fn, text):
        self.fn = fn
        self.text = text
        self.pos = Position(-1, 0, -1, fn, text)
        self.current_char = None
        self.advance()
```

Figure 3.3 Lexer Class

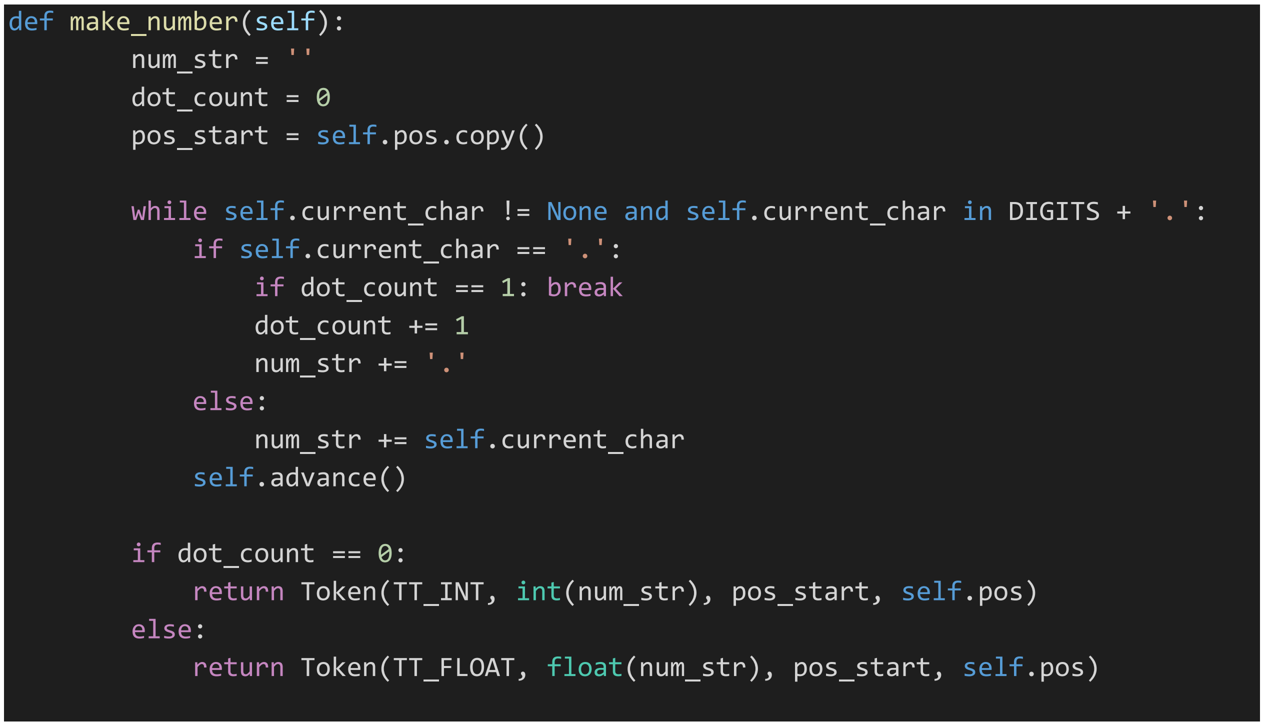
```
def advance(self):
        self.pos.advance(self.current_char)
        self.current_char = self.text[self.pos.idx] if self.pos.idx < len(self.te
xt) else None
```

Figure 3.4 Lexer Class, advance() function



```
def make_tokens(self):
        tokens = []

        while self.current_char != None:
            if self.current_char in ' \t':
                self.advance()
            elif self.current_char in DIGITS:
                tokens.append(self.make_number())
            elif self.current_char == '+':
                tokens.append(Token(TT_PLUS, pos_start=self.pos))
                self.advance()
            elif self.current_char == '-':
                tokens.append(Token(TT_MINUS, pos_start=self.pos))
                self.advance()
```

Figure 3.5 Lexer class, make_tokens() function



```
def make_number(self):
        num_str = ''
        dot_count = 0
        pos_start = self.pos.copy()

        while self.current_char != None and self.current_char in DIGITS + '.':
            if self.current_char == '.':
                if dot_count == 1: break
                dot_count += 1
                num_str += '.'
            else:
                num_str += self.current_char
            self.advance()

        if dot_count == 0:
            return Token(TT_INT, int(num_str), pos_start, self.pos)
        else:
            return Token(TT_FLOAT, float(num_str), pos_start, self.pos)
```

Figure 3.6 Lexer class, make_number() function

Parser

- After we've created a Lexer and performed our lexical analysis of the code, we need to organize the code.
- The Parser needs broken down code. It needs to make sense of the language construct like when to add, subtract, multiply, or divide.
- Simply put, our parser is a compiler that breaks the data into smaller elements from the Lexer and produces an output in form of a parse tree which our Interpreter will traverse.

How did we do it?

- We created a tree. A tree is data structure that consists of nodes that are organized into a hierarchy.
- For our programming language created an ABSTRACT SYNTAX TREE. Since our interpreter will be a syntax directed Programming language. The AST is a form of a parse tree.
- The parse tree will represent the syntax of our language and how they're prioritized.

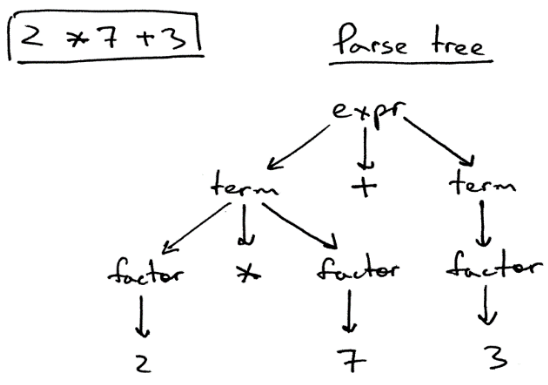


Figure 4.1 Parse Tree Representation, Source: https://ruslanspivak.com/lsbasi-part7/

- In this picture, the parse tree manages the sequence of rules the parser should apply to the input
- The root (top node) is labeled with the grammar start symbol
- Each lower node represents a rule application like expr, term, factor

Technically speaking, this method can really deter the performance and the speed of the compiler. Which is why we've used ASTs the performance issues of a standard parser tree.

- Instead of using a Parse Tree, we used something similar called the Abstract Syntax Tree.
- AST uses operators as root or interior nodes
- AST uses operands as children to root.
- ASTs are abstract they don't have to get into too much detail. Saves a lot of time.

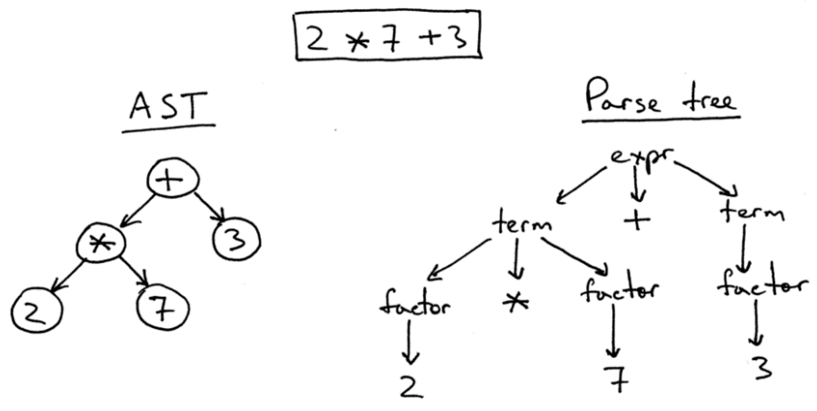


Figure 4.2 Parse Tree vs AST Representation, Source: https://ruslanspivak.com/lsbasi-part7/

WHY AST

OUR AST represents the syntax structure in abstract form. They're more compact They work similar to Parse Trees except they're much smaller. Example we if event A happens before B, we place A lower than B in the Tree.

In our AST we placed the operators with higher precedence lower in the tree so we can apply the BIDMAS/BODMAS rule.

- Instead of creating nodes for subtraction, addition, division, and multiplication, we just made a single node called BinOpNode. This node will represent every operator.
- To represent integers in our AST, we created a Number node that will define if the number is an Integer or a Float and return the value.
- In our code, each BinOP node will adopt the current value of node variable to the left and result of the call to the right. Pushing down nodes to the left.

Code

Node class

```python
class NumberNode:
    def __init__(self, tok):
        self.tok = tok

        self.pos_start = self.tok.pos_start
        self.pos_end = self.tok.pos_end

    def __repr__(self):
        return f'{self.tok}'
```

Figure 4.3 Node Class

Binary Operation Node Class

```python
lass BinOpNode:
    def __init__(self, left_node, op_tok, right_node):
        self.left_node = left_node
        self.op_tok = op_tok
        self.right_node = right_node

        self.pos_start = self.left_node.pos_start
        self.pos_end = self.right_node.pos_end

    def __repr__(self):
        return f'({self.left_node}, {self.op_tok}, {self.right_node})'
```

Figure 4.4 Binary Operation Class

Unary Operation Node Class

```python
class UnaryOpNode:
    def __init__(self, op_tok, node):
        self.op_tok = op_tok
        self.node = node

        self.pos_start = self.op_tok.pos_start
        self.pos_end = node.pos_end

    def __repr__(self):
        return f'({self.op_tok}, {self.node})'
```

Figure 4.5 Unary Operation Class

Parser Class

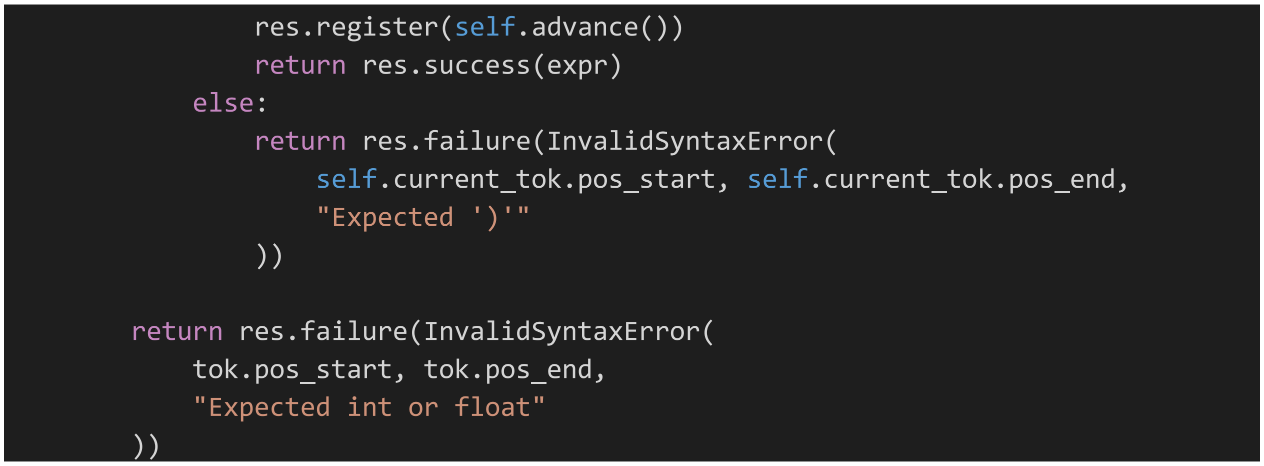
```python
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.tok_idx = -1
        self.advance()

    def advance(self, ):
        self.tok_idx += 1
        if self.tok_idx < len(self.tokens):
            self.current_tok = self.tokens[self.tok_idx]
        return self.current_tok
```

Figure 4.6 Parser class.

```python
 def parse(self):
        res = self.expr()
        if not res.error and self.current_tok.type != TT_EOF:
            return res.failure(InvalidSyntaxError(
                self.current_tok.pos_start, self.current_tok.pos_end,
                "Expected '+', '-', '*' or '/'"
            ))
        return res
```

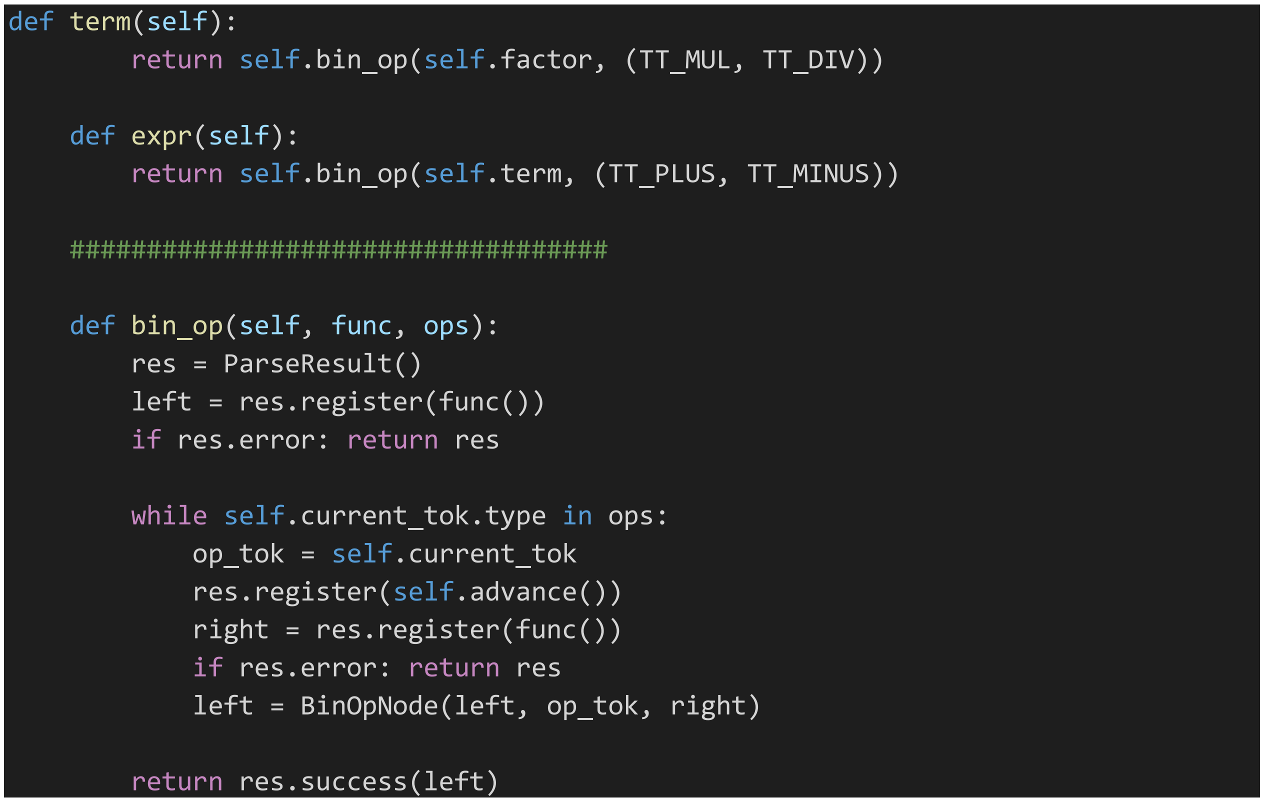
Figure 4.7 Parse Function

```python
def factor(self):
        res = ParseResult()
        tok = self.current_tok

        if tok.type in (TT_PLUS, TT_MINUS):
            res.register(self.advance())
            factor = res.register(self.factor())
            if res.error: return res
            return res.success(UnaryOpNode(tok, factor))

        elif tok.type in (TT_INT, TT_FLOAT):
            res.register(self.advance())
            return res.success(NumberNode(tok))

        elif tok.type == TT_LPAREN:
            res.register(self.advance())
            expr = res.register(self.expr())
            if res.error: return res
            if self.current_tok.type == TT_RPAREN:
```



```python
                res.register(self.advance())
                return res.success(expr)
            else:
                return res.failure(InvalidSyntaxError(
                    self.current_tok.pos_start, self.current_tok.pos_end,
                    "Expected ')'"
                ))

        return res.failure(InvalidSyntaxError(
            tok.pos_start, tok.pos_end,
            "Expected int or float"
        ))
```

Figure 4.8 Factor function

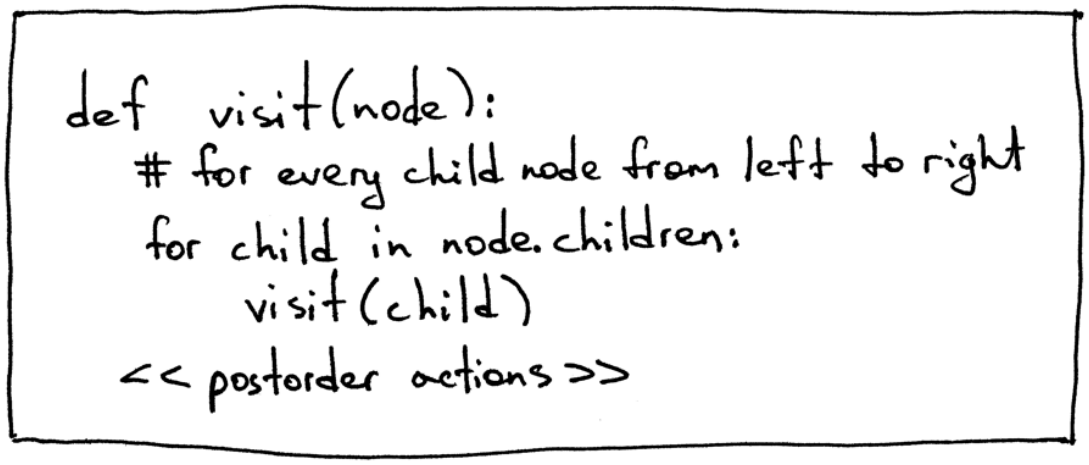


```python
def term(self):
        return self.bin_op(self.factor, (TT_MUL, TT_DIV))

    def expr(self):
        return self.bin_op(self.term, (TT_PLUS, TT_MINUS))

    ###################################

    def bin_op(self, func, ops):
        res = ParseResult()
        left = res.register(func())
        if res.error: return res

        while self.current_tok.type in ops:
            op_tok = self.current_tok
            res.register(self.advance())
            right = res.register(func())
            if res.error: return res
            left = BinOpNode(left, op_tok, right)

        return res.success(left)
```

Figure 4.9 Term Function, Expression Function, Binary Operation Function.

Interpreter

- After we created our Parser Tree / AST tree we need to navigate the tree so it can evaluate the expressions and execute the code.
- To do that we will be uses depth-first traversal
- In depth-first traversal we start at the root node and recursively visit the children of each node from left to right.



```
def visit(node):
    # for every child node from left to right
    for child in node.children:
        visit(child)
    << postorder actions >>
```

Figure 5.1 Pseudo Code for Interpreter

- The reason why we use DFT is because we need to evaluate the lower nodes because they represent operators with higher priority.
- The other reason is that we need to evaluate the operands of an operator before applying the operator to those operands.
- Our interpreter gets a parsed tree from the Parser and then simply traverses the tree to interpret the input.

Code

```python
class Interpreter:
    def visit(self, node, context):
        method_name = f'visit_{type(node).__name__}'
        method = getattr(self, method_name, self.no_visit_method)
        return method(node, context)

    def no_visit_method(self, node, context):
        raise Exception(f'No visit_{type(node).__name__} method defined')
```

Figure 5.2 Interpreter class.

```python
    def visit_NumberNode(self, node, context):
        return RTResult().success(
            Number(node.tok.value).set_context(context).set_pos(node.pos_start, n
ode.pos_end)
        )
```

Figure 5.3 Number node function.

```python
def visit_BinOpNode(self, node, context):
        res = RTResult()
        left = res.register(self.visit(node.left_node, context))
        if res.error: return res
        right = res.register(self.visit(node.right_node, context))
        if res.error: return res
        if node.op_tok.type == TT_PLUS:
            result, error = left.added_to(right)
        elif node.op_tok.type == TT_MINUS:
            result, error = left.subbed_by(right)
        elif node.op_tok.type == TT_MUL:
            result, error = left.multed_by(right)
        elif node.op_tok.type == TT_DIV:
            result, error = left.dived_by(right)
        if error:
            return res.failure(error)
        else:
            return res.success(result.set_pos(node.pos_start, node.pos_end))
```

Figure 5.4 BinaryOperationNode function.

```python
def visit_UnaryOpNode(self, node, context):
        res = RTResult()
        number = res.register(self.visit(node.node, context))
        if res.error: return res
        error = None
        if node.op_tok.type == TT_MINUS:
            number, error = number.multed_by(Number(-1))
        if error:
            return res.failure(error)
        else:
            return res.success(number.set_pos(node.pos_start, node.pos_end))
```

Figure 5.5 Unary Operation Node. After the Interpreter has run the functions in the Binary Operation Nodes, the Unary Operation node is run.

```
class Context:
    def __init__(self, display_name, parent=None, parent_entry_pos=None):
        self.display_name = display_name
        self.parent = parent
        self.parent_entry_pos = parent_entry_pos
```

Figure 5.6. Context class. When Traversing the Tree, we need to know where we are. Therefore, we've created a context class to track the parent node, visited nodes, unvisited nodes, and the child nodes.

```
class Number:
    def __init__(self, value):
        self.value = value
        self.set_pos()
        self.set_context()

    def set_pos(self, pos_start=None, pos_end=None):
        self.pos_start = pos_start
        self.pos_end = pos_end
        return self

    def set_context(self, context=None):
        self.context = context
        return self

    def added_to(self, other):
        if isinstance(other, Number):
          return Number(self.value + other.value).set_context(self.context), None
```

Figure 5.7 Number class. When the interpreter traverses the tree, it needs to run the corresponding functions in the code. Therefore, we've created a Number file that defines functions such as addition, subtraction, multiplication, and division.