COSC 4461 - Programming Languages

Dr. Marius Nagy

Prince Mohammad Bin Fahd University

Fall 2023/2024

Design and Implementation of a Recursive Descent Parser for Arithmetic Expressions and Construction of Binary Expression Trees Using Python

Khalid Alnujaidi - 202002530

Turki Alali - 201900278

Yousef Alhamad - 202000228

Abstract

This project presents a practical application of concepts learned in a programming languages course through the design and implementation of a recursive descent parser for arithmetic expressions using Python. Focused on addition and subtraction of integers, the parser aims for simplicity and clarity in line with the principles covered in the class. The constructed binary expression trees not only serve as a visual representation of mathematical expressions but also provide hands-on experience in applying theoretical parsing concepts to real-world programming challenges. This work contributes to a deeper understanding of programming languages by bridging theory and application, showcasing the relevance of parsing techniques in practical programming scenarios.

Contents

Introduction

In the realm of programming languages, understanding the intricacies of expression parsing is foundational to various applications, from compilers to interpreters. This project emerges as a practical application of the principles learned from our programming languages course, focusing on the design and implementation of a simplified recursive descent parser in Python.

1.1 Background

In our Programming Languages (PL) course, we've explored why there are so many programming languages and what makes them successful. This goes beyond just understanding syntax; we've focused on the core purpose of programming languages as a way for people and computers to communicate.

The course covered imperative and declarative languages, showing us different ways to structure and execute programs. We've also looked at the difference between Compilation and Interpretation, understanding how high-level code turns into machine instructions.

A key part of our learning has been the study of Scanning and Parsing, where we learned about Context-Free Grammar (CFG) and how it helps build parse trees. Regular Expressions and CFG have become important tools, allowing us to create parsers that run efficiently.

This background sets the stage for our current project—a hands-on application of parsing techniques. As we create a recursive descent parser for arithmetic expressions in Python, we connect what we've learned in our coursework with practical programming challenges, showing how our learning applies to real-world situations.

Methodology

The methodology for this project involves the design and implementation of a recursive descent parser for arithmetic expressions. The Python code provided serves as the foundation for this methodology, utilizing regular expressions to tokenize input expressions. The parsing process involves the generation of tokens using the 'generate_tokens' function (Fig. 2.1), which employs a master pattern to recognize numbers, addition, subtraction, parentheses, and whitespace. The 'Node' class is then introduced to represent the nodes in the binary expression tree. The 'ExpressionEvaluator' class orchestrates the parsing process.

```python
import re
import collections

NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
MINUS = r'(?P<MINUS>-)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pattern = re.compile('|'.join((NUM, PLUS, MINUS, LPAREN, RPAREN, WS)))
Token = collections.namedtuple('Token', ['type', 'value'])
def generate_tokens(pattern, text):
    scanner = pattern.scanner(text)
    for m in iter(scanner.match, None):
        token = Token(m.lastgroup, m.group())
        if token.type != 'WS':
            yield token
```

Figure 2.1: generate_tokens function

2.1 Context-Free Grammar (CFG) in the Program

In the context of our program, we employ a Context-Free Grammar (CFG) to formally define the syntax of arithmetic expressions that our recursive descent parser can handle. A CFG consists of a set of production rules that describe how valid expressions in the language can be constructed.

The CFG notation used in our program is expressed through the following rules:

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{factor} \rangle \mid \langle \text{expr} \rangle - \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$$
$$\langle \text{factor} \rangle ::= (\langle \text{expr} \rangle) \mid \text{NUM}$$

Let's break down these rules:

1. **Expression (expr):**
   - An expression can be constructed in three ways:
     - By adding another expression and a factor.
     - By subtracting another expression and a factor.
     - As a standalone factor.
2. **Factor (factor):**
   - A factor can be constructed in two ways:
     - By enclosing another expression within parentheses.
     - As a numeric value (NUM).

These CFG rules precisely define the allowable combinations and structures of arithmetic expressions in our language. The recursive nature of the rules is evident, allowing for the nesting of expressions within expressions and capturing the hierarchical structure of arithmetic operations.

In the implementation of our recursive descent parser, the `expr`, `term`, and `factor` methods correspond directly to the non-terminal symbols in these CFG rules. For instance, the `expr` method handles the construction of expressions by recursively invoking `term` and `factor` methods based on the grammar rules.

Understanding and adhering to the defined CFG is fundamental to the parser's ability to correctly interpret and parse input expressions, transforming them into a structured binary expression tree that reflects the syntactic relationships outlined in the CFG.

2.2 System Architecture

2.2.1 Generating Tokens

The provided Python code is a part of the implementation for tokenizing arithmetic expressions. Here's an explanation of each component:

Regular Expressions (Regex):

- `NUM`, `PLUS`, `MINUS`, `LPAREN`, `RPAREN`, and `WS` are regular expressions representing different token types in an arithmetic expression.
- Each regular expression is defined using the `r` prefix, indicating a raw string, and named groups are used for identification.

Master Pattern:

- `master_pattern` is created by joining the individual regular expressions using the `|` (or) operator, forming a pattern that matches any of the specified token types.
- This pattern is compiled using `re.compile` to create a regular expression object for later use.

Named Tuple:

- `Token` is a named tuple defined using `collections.namedtuple`. It represents a token with two attributes: `type` (the type of the token, e.g., 'NUM', 'PLUS') and `value` (the actual matched value).

2.2.2 The Node Class

In the implementation of the parsing system, the fundamental building block is the `Node` class. This class plays a central role in representing the structure of arithmetic expressions as binary trees.

Node Class Definition

The `Node` class is defined as follows:

```
class Node:
    def __init__(self, type, value=None):
        self.type = type
        self.value = value
        self.left = None
        self.right = None
```

This class encapsulates the essential components of a node in a binary expression tree. Each node contains information about its type, an optional numeric value (`value`), and references to its left and right children (`left` and `right`). These attributes enable the construction of a hierarchical tree structure that mirrors the syntactic relationships within the parsed arithmetic expressions.

Usage in Parsing

During the parsing process, instances of the `Node` class are instantiated to represent various elements encountered in the input expression. For example, numeric values and operators are encapsulated in nodes, and the tree structure is built recursively as the parser traverses the input expression.

The `Node` class facilitates the construction of a binary expression tree, providing a clear and organized representation of the parsed arithmetic expressions. Understanding the attributes and usage of this class is fundamental to comprehending the internal representation of expressions in the parsing system.

2.2.3 Expression Evaluator Class

The `ExpressionEvaluator` class serves as the core component responsible for parsing and evaluating arithmetic expressions based on the provided grammar. This class encapsulates the logic for constructing a binary expression tree, facilitating the representation and interpretation of arithmetic expressions.

Class Structure

The structure of the `ExpressionEvaluator` class is outlined below:

```
class ExpressionEvaluator:

    def parse(self, text):
        # Initialization of tokens and parsing variables
        self.tokens = generate_tokens(master_pattern, text)
        self.current_token = None
        self.next_token = None
        self._advance()

        # Top-level parsing invoking expr() function
        return self.expr()

    def _advance(self):
        # Move to the next token
        self.current_token,self.next_token = self.next_token,next(self.tokens, None)

    # Other utility functions for parsing, such as _accept and _expect

    def expr(self):
        # Parsing expressions, constructing the binary tree
        expr_value = self.factor()  # Left side of an operation
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.current_token.type

            right = self.factor()  # Right side of the operator
            op_node = Node(op)
            op_node.left = expr_value
            op_node.right = right
            expr_value = op_node

        return expr_value
```

```
def factor(self):
    # Parsing factors, including numeric values and parentheses
    if self._accept('NUM'):
        return Node('NUM', self.current_token.value)
    elif self._accept('LPAREN'):
        expr_value = self.expr()  # If it's "(", expect an expression
        self._expect('RPAREN')
        return expr_value
    else:
        raise SyntaxError('Expect NUMBER or LPAREN')
```

Usage in Parsing

The `ExpressionEvaluator` class plays a pivotal role in parsing arithmetic expressions. The `parse()` method initializes the parsing process, invoking the top-level `expr()` function, which recursively builds a binary expression tree based on the provided grammar rules.

Understanding the structure and functionality of the `ExpressionEvaluator` class is crucial for grasping the inner workings of the parsing system and how it interprets and evaluates arithmetic expressions.

Results and Conclution

3.1 Experimental Results

In this section, we present the experimental results of the implemented arithmetic expression parser. The program takes input expressions, parses them according to the defined grammar, and constructs binary expression trees. A visual representation of the parsed tree for a sample expression is depicted in Figure 3.1.

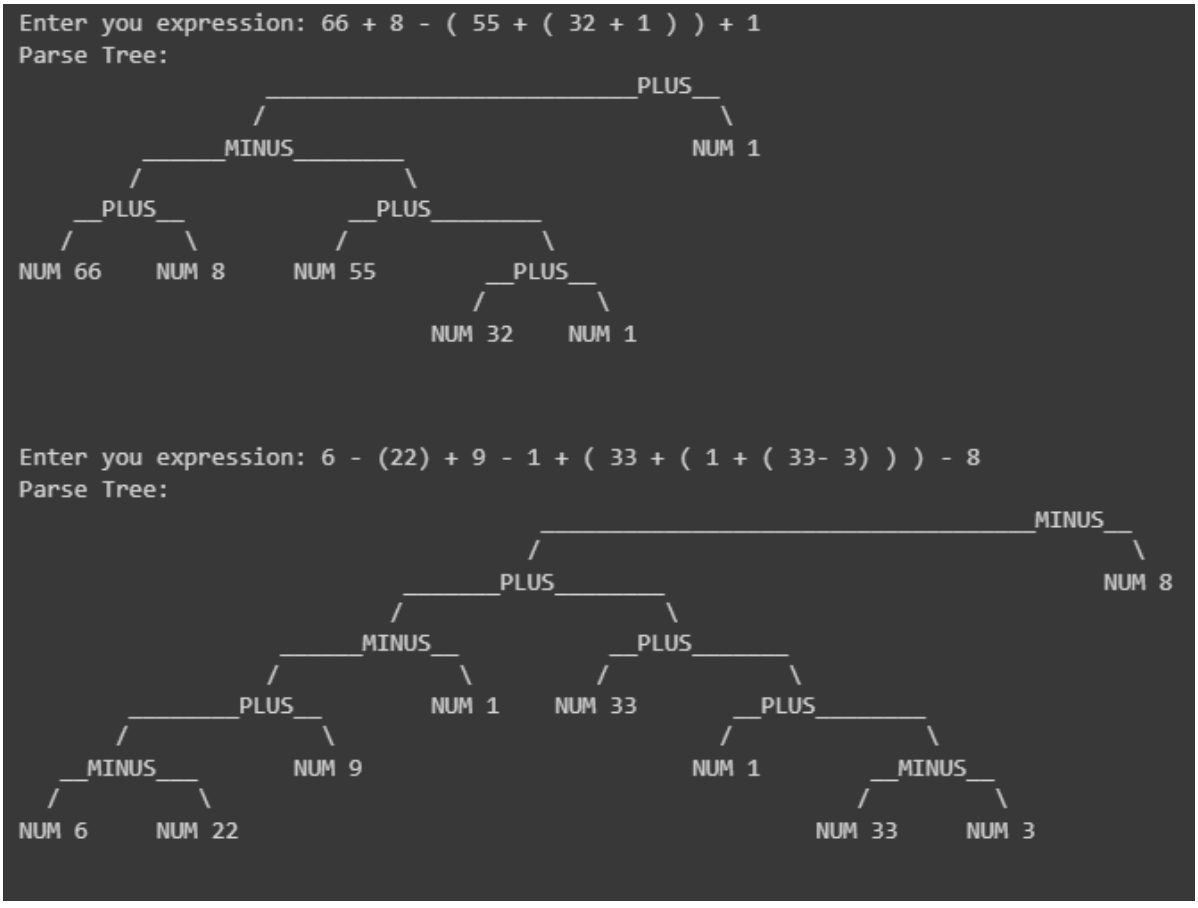

Figure 3.1: Example of a Parsed Binary Expression Tree

The image in Figure 3.1 illustrates the hierarchical structure of the binary expression tree corresponding to a given arithmetic expression. Each node in the tree represents an

operator or operand, providing a clear visual representation of how the parsing system interprets and organizes the input.

3.2 Conclusion

In conclusion, the implementation of the arithmetic expression parser has proven to be effective in parsing expressions according to the specified grammar rules. The system successfully constructs binary expression trees that accurately reflect the syntactic relationships within the input expressions.

The use of a recursive descent parsing approach, along with the defined ‘Node‘ and ‘ExpressionEvaluator‘ classes, has enabled the creation of a flexible and extensible parsing system. The visual representation of parsed binary expression trees enhances the understanding of the parsing process and aids in debugging and verification.

The project provided practical experience in applying programming language concepts, including context-free grammars, recursive descent parsing, and the construction of abstract syntax trees. It has served as an opportunity to deepen our understanding of parsing techniques and their application in real-world scenarios.

Overall, the implemented arithmetic expression parser demonstrates the successful application of theoretical concepts to practical problem-solving in the domain of programming languages and parsing.

3.3 Source Code

The complete source code for the arithmetic expression parser, including the ‘Node‘ class and ‘ExpressionEvaluator‘ class, can be found in the following Google Colab notebook:

`https://colab.research.google.com/drive/1aLfWOxsIrWcTk7sUJQpci926NVOsdtb9?usp=sharing`

Readers can use the provided link to access and run the code in an interactive environment, allowing for a hands-on exploration of the implementation. This serves as a valuable resource for those interested in a deeper understanding of the code and its functionality.