Laboratory Work No. 6

Course: Formal Languages & Finite Automata

Student: Iordan Liviu, Group: FAF-223

Topic: Parser & Building an Abstract Syntax Tree

Overview

The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example.

Similarly to a parse tree, in order to represent the structure of an input text one could create an Abstract Syntax Tree (AST). This is a data structure that is organized hierarchically in abstraction layers that represent the constructs or entities that form up the initial text. These can come in handy also in the analysis of programs or some processes involved in compilation.

Objectives

- ➔ Get familiar with parsing, what it is and how it can be programmed.
- ➔ Get familiar with the concept of AST.
- ➔ In addition to what has been done in the 3rd lab work do the following:
    - ◆ In case you didn't have a type that denotes the possible types of tokens you need to:

- Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
- Please use regular expressions to identify the type of the token.

◆ Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.

◆ Implement a simple parser program that could extract the syntactic information from the input text.

Introduction

Within the domain of formal languages and finite automata, the process of parsing and the construction of Abstract Syntax Trees (ASTs) serve as fundamental pillars underpinning the analysis and manipulation of textual data. Parsing, the systematic examination of text to discern its syntactic structure, lays the groundwork for various computational tasks, including compilation and program analysis. Similarly, ASTs offer a succinct and hierarchical representation of text, facilitating efficient program analysis and manipulation. In this laboratory work, we embark on a journey to delve into the intricacies of parsing and AST construction, aiming to deepen our understanding of these core concepts within the realm of formal language theory and computational linguistics.

Implementation description

The implementation of our parser and abstract syntax tree (AST) generation revolves around a cohesive system comprising several interconnected components. Here, we delineate the key aspects of our implementation, elucidating the functionality and interactions of each module:

1. Lexer Module (lexer.py):

- The lexer module encompasses the lexical analysis phase of our system, responsible for tokenizing the input text.
- Upon instantiation, the Lexer class initializes with the input text and necessary attributes to track the tokenization process.
- Tokenization is achieved through the tokenize() method, where the input text is scanned using regular expressions to identify tokens based on predefined token specifications.
- Each identified token is encapsulated within a Token object, containing its type and value, and subsequently appended to the list of tokens.

2. Abstract Syntax Tree (AST) Module (my_ast.py):

- The AST module defines the hierarchical structure representing the syntactic elements of the input text.
- Two primary classes, BinOp and Num, represent binary operations and numerical values, respectively, within the AST.
- Each BinOp instance encapsulates a left operand, operator, and right operand, forming a binary expression tree.
- Num instances represent numerical values extracted from the input text tokens.

3. Parser Module (parser.py):

- The parser module orchestrates the parsing process, converting the tokens generated by the lexer into a structured AST.
- The Parser class initializes with a lexer instance and orchestrates the parsing process through a series of methods corresponding to different syntactic elements.
- The parse() method serves as the entry point, initiating the parsing process by invoking the expression() method.

- Various parsing methods, including expression(), addition(), term(), and factor(), recursively analyze the token stream to construct the AST.
- Error handling mechanisms are implemented to detect and handle syntax errors encountered during parsing.

4. Main Program (main.py):

- The main program serves as the entry point for executing the parser and demonstrating its functionality.
- Input arithmetic expressions, featuring operators and parentheses, are provided for parsing and AST generation.
- Upon execution, the main program instantiates a lexer to tokenize the input text, followed by a parser to generate the AST.
- Finally, the tokens generated by the lexer and the resulting AST are printed to the console for inspection.

5. Token Type Enumeration (token_type.py):

- The TokenType enumeration defines distinct token types recognized during the lexical analysis phase.
- Each token type corresponds to a specific category of lexical elements, such as integers, operators, and parentheses.

By seamlessly integrating these components, our implementation achieves the overarching goal of parsing input arithmetic expressions and generating a structured AST representing their syntactic structure. Through modular design and systematic processing, our parser facilitates the analysis and manipulation of textual data within the domain of formal languages and finite automata.

Code Explanation

1. **lexer.py:**

```
from my_token import Token
from token_type import TokenType
import re



class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.current_token = None
        self.tokens = []

    def error(self):
        raise Exception('Invalid character')

    def tokenize(self):
        token_specification = [
            (TokenType.INTEGER, r'\d+'),
            (TokenType.PLUS, r'\+'),
            (TokenType.MINUS, r'\-'),
            (TokenType.TIMES, r'\*'),
            (TokenType.DIVIDE, r'\/'),
            (TokenType.LPAREN, r'\('),
            (TokenType.RPAREN, r'\)'),
            (TokenType.EOF, r'\Z')
        ]

        token_regex = '|'.join(f'(?P<{tok.name}>{pattern})' for tok, pattern in
token_specification)
        for mo in re.finditer(token_regex, self.text):
```

```
            kind = mo.lastgroup
            value = mo.group()
            tok_type = TokenType[kind]
            if tok_type == TokenType.INTEGER:
                value = int(value)  # Convert to integer
            self.tokens.append(Token(tok_type, value))

        self.tokens.append(Token(TokenType.EOF, None))
```

- Functionality: The Lexer class in lexer.py is responsible for tokenizing input text. It iterates through the input text and matches patterns corresponding to different token types using regular expressions.
- Implementation: The tokenize method utilizes regular expressions to identify tokens such as integers, arithmetic operators, and parentheses. It generates Token instances with the appropriate token type and value.
- Importance: The lexer is the initial stage in the syntactic analysis process, breaking down the input text into individual tokens. These tokens serve as the foundation for subsequent parsing and analysis tasks.

2. my_ast.py:

```
class AST:
    pass


class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right


class Num(AST):
    def __init__(self, token):
```

```
        self.token = token
        self.value = token.value



def print_ast(node, level=0):
    indent = '  ' * level
    if isinstance(node, BinOp):
        print(f'{indent}BinOp:')
        print(f'{indent}  Left:')
        print_ast(node.left, level+2)
        print(f'{indent}  Op: {node.op.value}')
        print(f'{indent}  Right:')
        print_ast(node.right, level+2)
    elif isinstance(node, Num):
        print(f'{indent}Num: {node.value}')
```

- Functionality: This file defines classes representing nodes in the Abstract Syntax Tree (AST) used for representing the syntactic structure of expressions.
- Implementation: Classes like AST, BinOp, and Num represent different types of nodes in the AST. The print_ast function recursively prints the AST structure for visualization and debugging purposes.
- Importance: The AST provides a structured representation of the syntactic elements of expressions. It abstracts away unnecessary details, facilitating subsequent analysis and manipulation tasks such as evaluation or code generation.

3. my_token.py:

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        return f'Token({self.type.name}, {repr(self.value)})'
```

- Functionality: The Token class defined in my_token.py represents individual tokens generated by the lexer. Each token has a type (defined in token_type.py) and a corresponding value.
- Implementation: The Token class encapsulates essential information about the lexical elements of the input text, providing a clear and structured representation of tokens.
- Importance: Tokens serve as the atomic units of the input text, providing well-defined elements for subsequent parsing and analysis. The Token class ensures consistency and clarity in representing lexical elements.

4. parser.py:

```
from token_type import *
from my_token import *
from my_ast import *



class Parser:
    def __init__(self, lexer):
        self.lexer = lexer
        self.tokens = lexer.tokens
        self.current_token = None
        self.pos = -1
        self.advance()

    def advance(self):
        self.pos += 1
        if self.pos < len(self.tokens):
            self.current_token = self.tokens[self.pos]
        else:
            self.current_token = Token(TokenType.EOF, None)

    def error(self):
```

```
        raise Exception('Invalid syntax')

    def parse(self):
        return self.expression()

    def expression(self):
        """ Parse an expression. """
        return self.addition()

    def addition(self):
        """ Handle addition and subtraction. """
        result = self.term()

        while self.current_token.type in (TokenType.PLUS, TokenType.MINUS):
            op = self.current_token
            self.advance()
            right = self.term()
            result = BinOp(left=result, op=op, right=right)

        return result

    def term(self):
        """ Handle multiplication and division. """
        result = self.factor()

        while self.current_token.type in (TokenType.TIMES, TokenType.DIVIDE):
            op = self.current_token
            self.advance()
            right = self.factor()
            result = BinOp(left=result, op=op, right=right)

        return result

    def factor(self):
        """ Handle parentheses and numbers. """
        token = self.current_token
```

```
        if token.type == TokenType.INTEGER:
            self.advance()
            return Num(token)
        elif token.type == TokenType.LPAREN:
            self.advance()
            result = self.expression()
            if self.current_token.type != TokenType.RPAREN:
                self.error()
            self.advance()
            return result
        else:
            self.error()
```

- Functionality: The Parser class in parser.py is responsible for parsing tokens generated by the lexer into an Abstract Syntax Tree (AST).
- Implementation: The Parser class implements methods for parsing expressions, handling addition, subtraction, multiplication, division, and parentheses. It constructs an AST representing the syntactic structure of expressions.
- Importance: By parsing tokens into an AST, the parser facilitates subsequent stages of analysis and manipulation of expressions. It enables tasks such as evaluation, optimization, or code generation based on the syntactic structure of expressions.

5. **main.py:**

```
from lexer import Lexer
from parser import Parser
from my_ast import print_ast


def main():
    # Input arithmetic expression with new operators and parentheses
    text = "(12 + 3) * 4 - 6 / (2 + 1)"
```

```
    # Create a lexer and tokenize the input text
    lexer = Lexer(text)
    lexer.tokenize()

    # Create a parser and parse the tokens into an AST
    parser = Parser(lexer)
    ast = parser.parse()

    # Print tokens generated by lexer
    print("Tokens:")
    for token in lexer.tokens:
        print(token)

    # Print AST generated by parser
    print("\nAST:")
    print_ast(ast)


if __name__ == "__main__":
    main()
```

- Functionality: main.py serves as the entry point for the program, orchestrating the process of lexical analysis (tokenization), syntactic analysis (parsing), and AST visualization.
- Implementation: The main function initializes a lexer, tokenizes input text, creates a parser, parses tokens into an AST, and finally visualizes the AST for debugging and visualization purposes.
- Importance: main.py provides a cohesive workflow for processing expressions, integrating the functionality of lexer, parser, and AST visualization. It serves as the central component for executing syntactic analysis tasks.

6. **token_type.py:**

```python
from enum import Enum



class TokenType(Enum):
   INTEGER = 'INTEGER'
   PLUS = 'PLUS'
   MINUS = 'MINUS'
   TIMES = 'TIMES'
   DIVIDE = 'DIVIDE'
   LPAREN = 'LPAREN'
   RPAREN = 'RPAREN'
   EOF = 'EOF'
```

- Functionality: token_type.py defines an enumeration TokenType representing different types of tokens that can be generated by the lexer.
- Implementation: Each token type corresponds to a specific lexical element such as INTEGER, PLUS, MINUS, etc.
- Importance: The TokenType enumeration provides a structured and standardized way to categorize tokens, facilitating the parsing process and subsequent stages of analysis. It ensures consistency in representing token types throughout the lexical analysis phase.

Conclusions:

In conclusion, the laboratory work focused on parsing and constructing Abstract Syntax Trees (ASTs) has provided invaluable insights into the foundational concepts within the domain of formal languages and finite automata. Through a systematic exploration of lexical analysis, syntactic parsing, and AST construction, we have gained a deeper understanding of the intricacies involved in processing textual data within computational frameworks.

The implementation and integration of a lexer, parser, and AST visualization component have demonstrated the practical application of theoretical concepts in real-world scenarios. By breaking down the input text into individual tokens, parsing them into hierarchical structures, and visualizing the abstract syntax, we have elucidated the step-by-step process involved in syntactic analysis.

Moreover, the laboratory work has underscored the importance of clear abstraction layers and modular design principles in software development. By delineating distinct components for lexical analysis, syntactic parsing, and AST construction, we have fostered a modular and extensible codebase, conducive to future enhancements and modifications.

Furthermore, the laboratory work has served as a platform for honing essential programming skills, including proficiency in Python programming, understanding of regular expressions, and familiarity with abstract data structures. These skills are not only pertinent within the realm of formal languages and finite automata but also find applications across diverse domains within computer science and software engineering.

In summary, the laboratory work has provided a comprehensive exploration of parsing and AST construction, equipping us with the theoretical knowledge and practical skills necessary to tackle complex computational tasks involving textual data. Through diligent inquiry, meticulous implementation, and rigorous documentation, we have embarked on a journey of discovery and learning, laying the foundation for future endeavors in computational linguistics and beyond.