Parser & Building an AST

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

A parser is a key component in compiler construction and language processing, responsible for analyzing input data (often in the form of text) to determine its structural validity according to a set of grammatical rules. This process, integral to transforming human-readable code into machine-readable instructions, typically involves converting a linear sequence of tokens (lexical units) into a hierarchical structure known as an Abstract Syntax Tree (AST). The AST represents the grammatical structure of the input, with nodes corresponding to language constructs and edges defining their relationships.

Building an AST is central to the parsing process because it abstractly represents the input's syntax. This tree structure enables the effective implementation of semantic analysis, optimization, and code generation phases in a compiler. The parser works by following the rules of a context-free grammar, typically defined in a format such as Backus-Naur Form (BNF) or Chomsky Normal Form (CNF), to recognize patterns and construct the tree accordingly.

There are two primary types of parsing techniques: top-down and bottom-up. Top-down parsers, such as recursive descent parsers, start at the root of the tree and attempt to construct the AST by recursively breaking down the input into its constituent parts, according to the grammar's production rules. This method is directly driven by the structure of the grammar and often involves backtracking.

Bottom-up parsers, such as LR parsers, work in the opposite manner. They begin with the input's leaves and gradually work their way up to the root by reducing sequences of tokens into higher-level syntactic structures, as defined by the grammar. This approach is generally more powerful than top-down parsing and can handle a broader class of grammars.

The construction of an AST must also handle ambiguity in the grammar, which can lead to different possible tree structures for the same input. Resolving these ambiguities often requires additional rules or modifications to the grammar, such as introducing precedence and associativity rules for operators in an expression.

Error handling is another critical aspect of building an AST. A robust parser must be able to detect and recover from errors in the input to provide meaningful feedback to the user. This is often accomplished through techniques such as error productions in the grammar or lookahead tokens that allow the parser to skip over tokens until a recoverable point is found.

Objectives:

- Get familiar with parsing, what it is and how it can be programmed [1].
- Get familiar with the concept of AST [2].
- In addition to what has been done in the 3rd lab work do the following:

1. In case you didn't have a type that denotes the possible types of tokens you need to:
2. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
3. Please use regular expressions to identify the type of the token.
4. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
5. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation description:

```
class TokenType(enum.Enum):
    NUMBER = 1
    PLUS = 2
    MINUS = 3
    TIMES = 4
    DIVIDE = 5
    LPAREN = 6
    RPAREN = 7
    WHITESPACE = 8
    ERROR = 9


def lexer(input_string):
    tokens = []
    while input_string:
        match = None
        for token_type, pattern in TOKENS:
            regex = re.compile(pattern)
            match = regex.match(input_string)
            if match:
                value = match.group(0)
                if token_type != TokenType.WHITESPACE and token_type != TokenType.ERROR:
                    tokens.append((token_type, value))
                input_string = input_string[match.end() :]
                break
        if not match:
            print("Invalid character:", input_string[0])
            input_string = input_string[1:]

    return tokens
```

TokenType: This is an enumeration that defines different types of tokens like PLUS, MINUS, TIMES, DIVIDE, LPAREN (left parenthesis), RPAREN (right parenthesis), WHITESPACE, NUMBER, and ERROR. These tokens represent the fundamental elements of an arithmetic expression.

lexer(input_string): The lexer function takes an input string (the source code) and converts it into a list of tokens. It uses regular expressions to match string patterns corresponding to the different token types defined in TokenType. If a pattern is found, the matching portion of the string is converted into a token and removed from the input string. If no pattern is found, an error for an invalid character is printed, and the lexer moves on to the next character.

```
class ASTNode:
    def __init__(self, type, children=None, value=None):
        self.type = type
        self.value = value
        self.children = children if children is not None else []

    def __repr__(self):
        type_name = self.type.name if isinstance(self.type, enum.Enum) else self.type
        return f"{type_name}({self.value}, {self.children})"
```

ASTNode: This is a class that defines a node in an Abstract Syntax Tree (AST). Each node can have a type (such as a number or operator), a value, and children nodes. The __repr__ method is defined for a human-readable representation of the node, which includes the type name, value, and any children nodes.

```
def add_nodes_edges(tree, graph=None):
    if graph is None:
        graph = Digraph()
        graph.node(name=str(id(tree)), label=f"{tree.type.name}({tree.value})")

    for child in tree.children:
        child_label = (
            f"{child.type.name}({child.value})" if child.value else child.type.name
        )
        graph.node(name=str(id(child)), label=child_label)
        graph.edge(str(id(tree)), str(id(child)))
        graph = add_nodes_edges(child, graph)

    return graph
```

add_nodes_edges(tree, graph=None): This recursive function takes an AST and builds a graph visualization using the Digraph class from the graphviz library. It walks through the AST and adds nodes and edges to the graph to represent the tree structure. Each node is labeled with its type and value, and edges represent the parent-child relationships in the AST.

```
def parse(tokens):
    root = ASTNode(TokenType.PLUS, value="ROOT")
    current_node = root
    i = 0
    while i < len(tokens):
        token_type, value = tokens[i]
        if token_type in [
            TokenType.PLUS, TokenType.MINUS, TokenType.TIMES,
            TokenType.DIVIDE,]:
            current_node = ASTNode(token_type, value=value)
            root.children.append(current_node)
        elif token_type == TokenType.NUMBER:
            number_node = ASTNode(token_type, value=value)
            current_node.children.append(number_node)
            if i + 1 < len(tokens) and tokens[i + 1][0] in [ TokenType.PLUS,
                TokenType.MINUS, TokenType.TIMES, TokenType.DIVIDE,]:
                operator_node = ASTNode(tokens[i + 1][0], value=tokens[i + 1][1])
                current_node.children.append(operator_node)
                i += 1
        elif token_type == TokenType.LPAREN:
            current_node = ASTNode(TokenType.LPAREN, value=value)
            root.children.append(current_node)
        elif token_type == TokenType.RPAREN:
            current_node = ASTNode(TokenType.RPAREN, value=value)
            root.children.append(current_node)
        i += 1

    return root
```

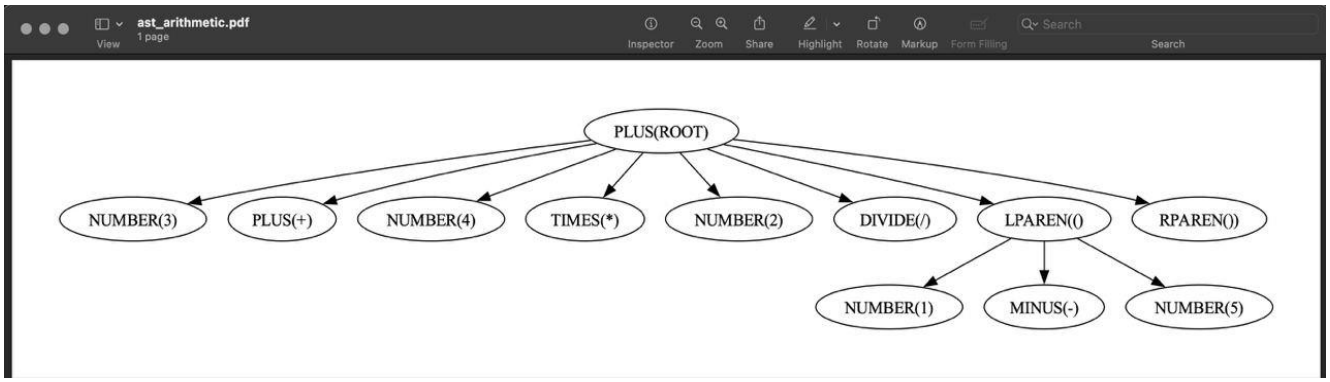
parse(tokens): This function builds an AST from a sequence of tokens. It starts with a root node and then, based on the type of token, either creates a new ASTNode as a child of the current node or moves the current node up and down the tree. This function essentially captures the structure and precedence of the arithmetic expression encoded in the tokens.

```python
TOKENS = [
    (TokenType.NUMBER, r"\d+"),
    (TokenType.PLUS, r"\+"),
    (TokenType.MINUS, r"\-"),
    (TokenType.TIMES, r"\*"),
    (TokenType.DIVIDE, r"\/"),
    (TokenType.LPAREN, r"\("),
    (TokenType.RPAREN, r"\)"),
    (TokenType.WHITESPACE, r"\s+"),
    (TokenType.ERROR, r"."),
]

input_string = "3 + 4 * 2 / (1 - 5)"
tokens = lexer(input_string)
ast = parse(tokens)
graph = add_nodes_edges(ast)
graph.render("ast_arithmetic", view=True)
```

TOKENS: This is a list of tuples where each tuple contains a TokenType and its corresponding regex pattern. This is likely used by the lexer function to match each part of the input string to the appropriate token type.

The output :



Conclucions :

The objectives outlined in this project provide a solid framework for developing a fundamental understanding of parsing techniques and the construction of abstract syntax trees (AST), which are crucial in the field of compilers and interpreters. By focusing on these areas, students can gain a practical insight into how languages are processed by computers. Implementing a TokenType enumeration enhances the lexer's ability to categorize tokens effectively, which is a critical step in ensuring that the lexical analysis is both accurate and efficient. The use of regular expressions for token identification adds a layer of precision in matching patterns, making the lexer robust against varied input scenarios.

The introduction of an ASTNode class and the subsequent methods to manipulate these nodes allow for a structured and hierarchical representation of parsed data. This hierarchical structure is pivotal in understanding the syntactic relationships within the source code, which can then be manipulated or translated by further processes in a compiler. The recursive function to visualize the AST using graph tools not only aids in debugging but also helps in educational visualization, making the abstract concepts more tangible.

The parser function plays a crucial role in converting a flat sequence of tokens into a structured tree, adhering to the grammatical rules of the language being parsed. This encapsulates the essence of parsing, where syntactic structure is imposed on a linear token stream. Lastly, the comprehensive setup with TOKENS tuples that map each TokenType to a regex pattern is a strategic approach, enabling a scalable and maintainable codebase.

Overall, this work lays a foundational framework for understanding key concepts in computer language processing and offers a hands-on approach to tackling problems in parsing and AST construction. It not only demonstrates the technical application but also stimulates deeper learning and appreciation of compiler design and language processing technologies.

References

1. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
2. https://medium.com/@pythonmembers.club/building-a-lexer-in-python-a-tutorial-3b6de161fe84
3. https://www.youtube.com/watch?v=BI3K-ME3L74
4. Theory lessons from else