MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

Laboratory work 6:
Parser and Building an Abstract Syntax Tree.

Course: Formal Languages and Finite Automata

Author: Cucoș Maria

Chișinău, 2024

TABLE OF CONTENTS

Theory

Parsers and ASTs play a crucial role in language processing and enable a wide range of language-related tasks, from parsing and analysis to code generation and optimization. The following two subsections cover the theory for parsers and Abstract Syntax Trees.

Parsers

Parsers are software components or tools that analyze the syntax of a given input according to a specified grammar. They take input in the form of a sequence of tokens and determine whether the input conforms to the syntax rules defined by the grammar. Parsers are commonly used in programming language compilers, interpreters, and other language processing tools. There are different types of parsers, including:

1. **Recursive Descent Parser:** A top-down parser that starts from the top-level grammar rule and recursively expands non-terminal symbols until the entire input is parsed.
2. **LL Parser (Left-to-Right, Leftmost Derivation):** Another type of top-down parser that reads input from left to right and constructs a leftmost derivation of the input string.
3. **LR Parser (Left-to-Right, Rightmost Derivation):** A bottom-up parser that constructs a rightmost derivation of the input string by repeatedly applying reduction steps to the input.
4. **Parser Combinators:** A technique for building parsers by combining simpler parsing functions or combinators to create more complex parsers.
5. **PEG Parser (Parsing Expression Grammar):** A type of parser that allows for more flexibility in defining grammars compared to traditional context-free grammars.

Abstract Syntax Trees (ASTs)

An Abstract Syntax Tree (AST) is a tree-like data structure that represents the syntactic structure of a program or expression. It captures the hierarchical relationship between the different components of the code, such as statements, expressions, and operators, while abstracting away details such as whitespace and syntax tokens. ASTs are commonly used in compilers and interpreters to facilitate various language processing tasks, including:

1. **Semantic Analysis:** ASTs are used to perform semantic analysis of the code, such as type checking and scope resolution.
2. **Code Generation:** ASTs serve as an intermediate representation of the code during the code generation phase of compilation, where machine-readable code is produced from the parsed source code.
3. **Optimization:** ASTs can be manipulated and transformed to apply various optimization techniques to the code, such as constant folding and dead code elimination.
4. **Language Tooling:** ASTs are used in language tooling and IDEs to provide features such as syntax highlighting, code completion, and refactoring.

Objectives

1. Get familiar with parsing, what it is and how it can be programmed.
2. Get familiar with the concept of AST.
3. In addition to what has been done in the 3rd lab work do the following:
   (a) In case you didn't have a type that denotes the possible types of tokens you need to:
      i. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
      ii. Use regular expressions to identify the type of the token.
   (b) Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
   (c) Implement a simple parser program that could extract the syntactic information from the input text.

Implementation Description

Parser

I implemented the parser logic in a `Parser` class. I started by importing the `Lexer` class from the `lexer` module and all classes (`Node, SymbolNode, VariableNode, StringNode`) from the `ast_1` module. After that, in the `Parser` class I define the `parse` method that takes a list of lexemes and parses them into instructions. It iterates over each line of lexemes, extracts the instruction type and its arguments, and stores them as a tuple in the `instructions` list.

```
def parse(self, lexemes_list):
    for line_lexemes in lexemes_list:
        if len(line_lexemes) < 2:
            continue
        instruction = line_lexemes[0][1]
        arguments = []
        for token in line_lexemes[1:]:
            if token[0] == "Variable":
                arguments.append(("Variable", token[1]))
            elif token[0] == "String":
                arguments.append(("String", token[1]))
        self.instructions.append((instruction, arguments))
```

The `get_instructions` method returns the parsed instructions stored in the `instructions` list.

```
    def get_instructions(self):
        return self.instructions
```

The `main` function instantiates a `Lexer` and a `Parser`. It prompts the user to enter commands and reads input until an empty line is entered. The input is tokenized using the lexer, parsed using the parser, and the parsed instructions are printed.

```
if __name__ == "__main__":
    lexer = Lexer()
    parser = Parser()
    print("Enter your commands (press Enter after each command):")
    inputs = []
    while True:
        line = input().strip()
        if not line:
            break
        inputs.append(line)
    lexemes_list = lexer.tokenize("\n".join(inputs))
    parser.parse(lexemes_list)
    instructions = parser.get_instructions()
    for instruction in instructions:
        print("Parsed command:", instruction)
```

Abstract Syntax Trees

`Node` represents a generic node in the abstract syntax tree (AST). It has an attribute value to store the value of the node and a list children to store its child nodes.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child):
        self.children.append(child)
```

`SymbolNode`, `VariableNode`, `StringNode`are subclasses of `Node` representing specific types of

nodes in the AST. They override the `__str__` method to provide a string representation of the node.

```
class SymbolNode(Node):
    def __str__(self):
        return f"<SymbolNode {self.value}>"

class VariableNode(Node):
    def __str__(self):
        return f"<VariableNode {self.value}>"

class StringNode(Node):
    def __str__(self):
        return f"<StringNode {self.value}>"
```

`construct_ast` takes a list of parsed commands and constructs an AST based on those commands. It iterates over each parsed command, creates nodes for the symbol and its details (variable and string), links the nodes together, and sets the root node of the AST. Finally, it returns it.

```
def construct_ast(parsed_commands):
    root = None
    for command in parsed_commands:
        symbol, details = command
        symbol_node = SymbolNode(symbol)
        variable_node = VariableNode(details[0][1])
        string_node = StringNode(details[1][1])
        symbol_node.add_child(variable_node)
        variable_node.add_child(string_node)
        if root is None:
            root = symbol_node
    return root
```

Testing

By executing the `parser` file, we get the following output. This output demonstrates the process of parsing commands and constructing an Abstract Syntax Tree (AST) for each command. The user is prompted to enter commands, one at a time. After entering each command, they press Enter to input the next one. After parsing each command, the parser outputs the parsed command as a tuple. Following the parsed command, the output displays the corresponding AST for that command. Each AST is represented

hierarchically, with the symbol node at the top, followed by the variable node, and then the string node.

```
Enter your commands (press Enter after each command):
detect imagePath="path/to/image1.png"
recognize imagePath="path/to/image2.png"

Parsed command: ('detect', [('Variable', 'imagePath'), ('String', '"path/to/image1.png"')])
AST:
<SymbolNode detect>
└── <VariableNode imagePath>
    └── <StringNode "path/to/image1.png">
Parsed command: ('recognize', [('Variable', 'imagePath'), ('String', '"path/to/image2.png"')])
AST:
<SymbolNode recognize>
└── <VariableNode imagePath>
    └── <StringNode "path/to/image2.png">
```

Conclusion

In conclusion, this report has presented an implementation of a parser and Abstract Syntax Tree (AST) generator for a simple command language. The parser takes input commands, tokenizes them, and extracts the necessary information to construct an AST representing the hierarchical structure of each command. The parser implementation involves several key components, including the Lexer, which tokenizes input commands, and the Parser, which parses the tokens and constructs the ASTs. The AST is represented using custom Node classes, including SymbolNode, VariableNode, and StringNode, each encapsulating relevant information about the command structure. Through the provided code examples and output demonstrations, we've showcased the functionality and effectiveness of the parser and AST generator. The parser successfully handles input commands, parses them into structured data, and constructs ASTs that accurately represent the commands' hierarchical relationships.

Overall, this implementation serves as a foundational framework for processing commands in a domain-specific language (DSL), demonstrating the essential steps involved in parsing and representing commands using ASTs. Future enhancements could include adding support for more complex commands, error handling, and optimizing performance for larger command sets.