Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 6
Discipline: Formal Languages & Finite Automata
Topic: Parser & Building an Abstract Syntax Tree.

Done by: Barbarov Nadejda st.gr., FAF-221

Checked by:
asist.univ. Crețu Dumitru

Chișinău 2024

Theory

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:

1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST [2].
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. Please 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

First of all, this line imports the regular expression module (re), which provides functionality for working with regular expressions in Python. It's utilized later in the program to match patterns for token identification:

```
import re
```

Next, token types are defined as raw strings (TOKEN_INT, TOKEN_PLUS, etc.), serving as identifiers for different types of tokens in the input expression. Regular expression patterns are associated with each token type in the patterns list. These patterns are used by the lexer to recognize and tokenize the input text.

```
# Define token types
TOKEN_INT = r'INT'
TOKEN_PLUS = r'PLUS'
TOKEN_MINUS = r'MINUS'
TOKEN_MULTIPLY = r'MULTIPLY'
TOKEN_DIVIDE = r'DIVIDE'
TOKEN_LPAREN = r'LPAREN'
TOKEN_RPAREN = r'RPAREN'
```

Figure 1. Token Types definition

The Token class encapsulates individual tokens, storing information about their type and value. Its __str__ method provides a string representation of the token, useful for debugging and displaying token information.

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

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

Figure 2. Token class

The next list contains tuples where each tuple consists of a token type and its corresponding regular expression pattern. Each pattern is used by the lexer to recognize specific token types in the input text.

```
# Define patterns for token identification
patterns = [
    (TOKEN_INT, r'\d+'),
    (TOKEN_PLUS, r'\+'),
    (TOKEN_MINUS, r'\-'),
    (TOKEN_MULTIPLY, r'\*'),
    (TOKEN_DIVIDE, r'/'),
    (TOKEN_LPAREN, r'\('),
    (TOKEN_RPAREN, r'\)'),
]
```

Figure 3. Pattern list

The Lexer class is responsible for tokenizing input text. It iterates through the input text and matches patterns defined earlier to identify tokens. The get_next_token method retrieves the next token from the input text using regular expression pattern matching.

```
class Lexer:
    Nadejda Barbarov
    def __init__(self, text):
        self.text = text
        self.pos = 0

    1 usage  Nadejda Barbarov
    def error(self):
        raise Exception('Invalid character')

    1 usage  Nadejda Barbarov
    def get_next_token(self):
        while self.pos < len(self.text):
            current_text = self.text[self.pos:]
            for token_type, pattern in patterns:
                match = re.match(pattern, current_text)
                if match:
                    value = match.group(0)
                    self.pos += len(value)
                    return Token(token_type, value)

            # If no match found, raise an error
            self.error()

        return Token( type: None,  value: None)
```

Figure 4. Lexer class

ASTNode, BinOpNode, and NumNode are classes representing different types of nodes in the Abstract Syntax Tree (AST) that represents the structure of the parsed expression. BinOpNode represents binary operations (e.g., addition, subtraction), NumNode represents numeric values, and ASTNode serves as a base class for all AST nodes.

```
class ASTNode:
    pass

2 usages  Nadejda Barbarov
class BinOpNode(ASTNode):
    Nadejda Barbarov
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

    Nadejda Barbarov
    def __str__(self):
        return f'({self.left} {self.op} {self.right})'

1 usage  Nadejda Barbarov
class NumNode(ASTNode):
    Nadejda Barbarov
    def __init__(self, token):
        self.token = token
        self.value = token.value

    Nadejda Barbarov
    def __str__(self):
        return str(self.value)
```

Figure 5. AST classes

The **Parser** class processes the tokens generated by the lexer and constructs an AST that reflects the syntactic structure of the expression. Methods such as **advance**, **expr**, **term**, and **factor** handle different parts of the parsing process, including advancing through tokens, parsing expressions, terms, and factors, and building the AST nodes accordingly.

The **main** function serves as the entry point of the program. It prompts the user to input an expression, then utilizes the lexer and parser to tokenize and parse the expression, respectively. Finally, it displays the tokens extracted from the input expression and the resulting AST for visual inspection and verification.

Conclusions / Screenshots / Results

In summary, this laboratory work provided a practical exploration of language processing concepts, including lexical analysis, parsing, and abstract syntax trees (ASTs). Through the implementation of a lexer, parser, and AST node classes, we gained insights into tokenization techniques, recursive descent parsing, and modular program design. This hands-on experience deepened our understanding of language processing fundamentals and their application in building language processing systems.

```
Enter an expression: 2+14/7
Tokens:
Token(INT, 2)
Token(PLUS, +)
Token(INT, 14)
Token(DIVIDE, /)
Token(INT, 7)

AST:
(2 + (14 / 7))
```

Figure 6. End result

References

1. **Wikipedia.** https://en.wikipedia.org/wiki/Parsing
2. **Wikipedia**. https://en.wikipedia.org/wiki/Abstract_syntax_tree
3. **Github.**https://github.com/filpatterson/DSL_laboratory_works/tree/master/6_ParserASTBuild
4. **Github.** https://github.com/nadea-b/LFA