**Ministerul Educaţiei și Cercetării al Republicii Moldova**
**Universitatea Tehnică a Moldovei**
**Facultatea Calculatoare, Informatică și Microelectronică**

# Laboratory work 6:
# Course: Formal Languages and Finite Automata
# Topic: Parser & Building an Abstract Syntax Tree

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

## TABLE OF CONTENTS

## THEORY

A parser is a crucial component of a compiler or interpreter that analyzes the syntactic structure of a program to ensure it follows the grammar rules of a programming language. The parser reads the program's source code and produces a data structure called the Abstract Syntax Tree (AST), which represents the program's syntactic structure in a hierarchical manner.

**Abstract Syntax Tree (AST)**: An AST is a tree representation of the syntactic structure of a program, where each node represents a syntactic construct (e.g., expressions, statements, declarations). The tree's structure reflects the hierarchy of the language's grammar rules, with parent nodes representing higher-level constructs and child nodes representing their components.

**Parsing Techniques**:

1. **Recursive Descent Parsing**: A top-down parsing technique where each non-terminal symbol in the grammar is associated with a parsing function. The parser recursively calls these functions to match the input tokens with the grammar rules.
2. **LR Parsing**: A bottom-up parsing technique that builds the parse tree from leaves to the root. LR parsers use a parsing table generated from the grammar to guide the parsing process.

**Building an Abstract Syntax Tree (AST)**:

1. **Lexer (Tokenizer)**: The first step is to tokenize the input source code into a sequence of tokens, representing the smallest syntactic units of the language (e.g., keywords, identifiers, operators).
2. **Parser**: The parser reads the sequence of tokens and constructs the AST according to the grammar rules. During parsing, the parser applies syntactic analysis techniques to ensure the input adheres to the language's syntax.
3. **AST Construction**: As the parser recognizes syntactic constructs, it constructs nodes for the AST. Each node typically contains information about the syntactic category it represents and references to its children nodes, reflecting the hierarchical structure of the program.

## 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:
    i. 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.
    ii. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
    iii. Implement a simple parser program that could extract the syntactic information from the input text.

## IMPLEMENTATION

The provided Python code consists of two files: Lab_6.py → implements a lexer, parser, and Abstract Syntax Tree (AST) builder for a simple arithmetic expression language and test.txt → where to test different expresions.

### Lab_6.py:

### Lexer (Tokenizer):

The lexer function tokenizes the input text into tokens based on predefined regular expressions for different token types. It iterates through the input text, attempting to match each token pattern using regular expressions. When a match is found, it yields a Token object containing the token type and value. Whitespace tokens are ignored. The lexer utilizes regular expressions defined in token_patterns to match different token types, such as integers, floats, operators, and parentheses.

```
def lexer(text):

    pos = 0

    tokens = []

    while pos < len(text):

        for token_type, pattern in token_patterns:

            regex = re.compile(pattern)

            match = regex.match(text, pos)

            if match:

                value = match.group(0)

                pos = match.end()

                if token_type != TokenType.WHITESPACE:

                    yield Token(token_type, value)

                break

        else:

            raise Exception("Invalid character")
```

**Parser:**

The parser function parses the sequence of tokens generated by the lexer and constructs an AST representing the syntactic structure of the input expression. It utilizes a stack-based approach to handle operator precedence and parentheses. The parser iterates through the tokens, constructing nodes for the AST based on the token types and their relationships. The parser uses a stack (brackets_stack) to handle nested parentheses and maintain the hierarchy of sub-expressions.

```
def parser(tokens):

    root = Node("expression")

    current_node = root

    brackets_stack = []

    stack_len = 0

    flag = False

    for token in tokens:

        if token.type == TokenType.LPAREN:

            current_node = Node("sub_expression", parent=current_node)

            brackets_stack.append(current_node)

            ...
```

**AST Construction:**

During parsing, the parser constructs nodes for the AST representing operands, operators, and sub-expressions. Nodes are created for each token, and their parent-child relationships are established based on the expression's structure.

```
def parser(tokens):

    ...

    for token in tokens:

        ...

        elif token.type == TokenType.MULTIPLY:

            last_node = current_node.children[-1]

            last_node.parent = None

            current_node = Node("multiply", parent=current_node)

            last_node.parent = current_node

            ...
```

**Printing the AST:**

The print_tree function prints the AST in a hierarchical format using the RenderTree utility from the anytree library. It traverses the AST in pre-order and prints each node along with its parent-child relationships.

```
def print_tree(node):

    for pre, fill, node in RenderTree(node):

        print("%s%s" % (pre, node.name))
```

**Execution:**

The main function reads the input text from a file named test.txt. It tokenizes the text using the lexer, parses the tokens to build the AST, and finally prints the AST using the print_tree function.

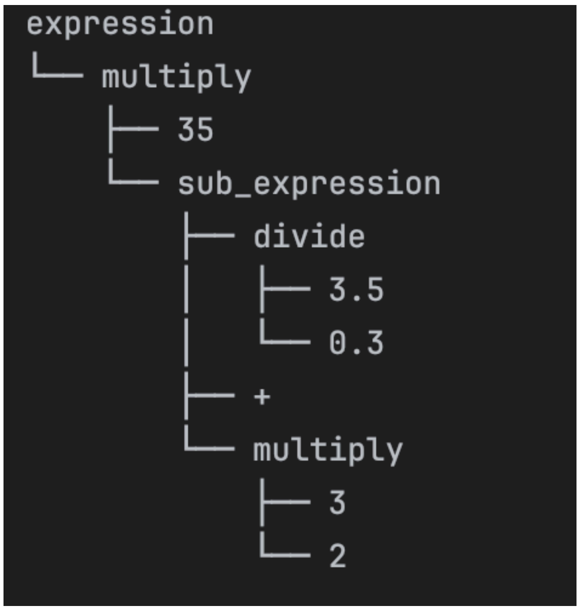
```
def main():

    file = open("test.txt", "r")

    text = file.read()

    file.close()

    tokens = list(lexer(text))

    root = parser(tokens)

    print_tree(root)


if __name__ == '__main__':

    main()
```

**test.txt:**

The input text file contains an arithmetic expression, which serves as the test input for the lexer, parser, and AST builder.

```
35 * (3.5 / 0.3 + 3 * 2)
```

## RESULTS



```
expression
└── multiply
    ├── 35
    └── sub_expression
        ├── divide
        │   ├── 3.5
        │   └── 0.3
        ├── +
        └── multiply
            ├── 3
            └── 2
```

*Figure 1 Lab_6 Result*

## CONCLUSION

In this laboratory work, we explored the concepts of parsing and Abstract Syntax Trees (ASTs) within the context of formal languages and finite automata. By implementing a lexer, parser, and AST builder for a simple arithmetic expression language, we gained practical experience in syntactic analysis and AST construction.

The lexer efficiently tokenizes input text using regular expressions to categorize tokens into predefined types, facilitating further syntactic analysis. It provides a foundational step in the parsing process by breaking down the input into manageable units.

The parser utilizes a stack-based approach to parse tokens generated by the lexer and construct an AST representing the syntactic structure of the input expression. By recursively traversing the tokens and applying grammar rules, the parser builds a hierarchical tree structure that captures the relationships between different syntactic elements.

The construction of the AST reflects the hierarchical nature of the input expression, with nodes representing operands, operators, and sub-expressions. This hierarchical representation facilitates subsequent analysis and manipulation of the input expression.

Through the implementation of the lexer, parser, and AST builder, we gained insights into fundamental parsing techniques and AST construction principles. These concepts are essential in the development of compilers, interpreters, and other language processing tools.

Overall, this laboratory work enhanced understanding of formal languages and finite automata, equipping us with valuable skills in lexical and syntactic analysis.

## REFERENCES

https://github.com/Chiuliana/DSL_Laboratory_Works/tree/main