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

Laboratory work nr. 6
Course: Formal languages and finite automata
Topic: Parser & Building an Abstract Syntax Tree

Elaborated:
st. gr. FAF-223 Pereteatcu Arina

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

During the syntactic analysis stage, parsing takes place. A stream of tokens is received from the lexical analyzer, and the parser generates a parser tree for each token while comparing it to the syntactic faults.

A compiler determines whether or not the tokens produced by the lexical analyzer are arranged in accordance with the language's syntactic rules during the syntax analysis stage. This is what a parser does. The lexical analyzer provides a string of tokens to the parser, which then confirms that the string may represent the grammar of the source language. It creates a parse tree from which intermediate code can be written, and it finds and indicates any syntactic mistakes.
Types of Parsing:

1. Top-down Parsing

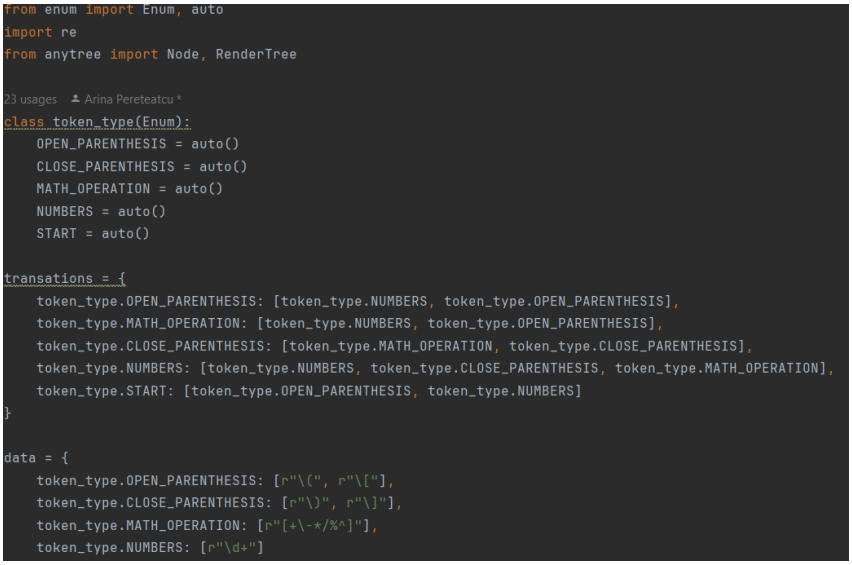
2. Bottom-up Parsing

Objectives:

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

Implementation Description

I wrote the code for this laboratory work in Python.



```python
from enum import Enum, auto
import re
from anytree import Node, RenderTree

23 usages  Arina Pereteatcu *
class token_type(Enum):
    OPEN_PARENTHESIS = auto()
    CLOSE_PARENTHESIS = auto()
    MATH_OPERATION = auto()
    NUMBERS = auto()
    START = auto()

transations = {
    token_type.OPEN_PARENTHESIS: [token_type.NUMBERS, token_type.OPEN_PARENTHESIS],
    token_type.MATH_OPERATION: [token_type.NUMBERS, token_type.OPEN_PARENTHESIS],
    token_type.CLOSE_PARENTHESIS: [token_type.MATH_OPERATION, token_type.CLOSE_PARENTHESIS],
    token_type.NUMBERS: [token_type.NUMBERS, token_type.CLOSE_PARENTHESIS, token_type.MATH_OPERATION],
    token_type.START: [token_type.OPEN_PARENTHESIS, token_type.NUMBERS]
}

data = {
    token_type.OPEN_PARENTHESIS: [r"\(", r"\["],
    token_type.CLOSE_PARENTHESIS: [r"\)", r"\]"],
    token_type.MATH_OPERATION: [r"[+\-*/%^]"],
    token_type.NUMBERS: [r"\d+"]
```

Figure1 .Class Token Type

This is a Python script that describes the types of tokens for a basic lexer, as well as regular expressions to match tokens of each type and transitions between these types.
The script imports the required modules, such as Node, RenderTree from the anytree module, and

Enum from the enum module, as well as re for regular expressions. The Python enumeration classes can be created using the Enum module. You can create symbolic names for a collection of distinct values using enumerations, which improves the readability and maintainability of your code. The token_type enumeration class, which represents various token kinds, is defined in this script using Enum. Python support for manipulating tree data structures is provided by the anytree package. In particular, it makes it simple to build, navigate, and work with tree structures. Nodes are nodes in trees, and by making instances of the Node class and connecting them, you can build hierarchical structures. Anytree offers a tool called RenderTree that allows you to see the tree structure in a legible way. It enables you to display the node hierarchy of a tree as a string. For managing regular expressions, Python comes with a built-in regular expression handling library called re.

Regular expressions are effective tools for manipulating strings and matching patterns. They facilitate the identification and extraction of certain textual segments by enabling the specification of intricate search strategies. Regular expressions are defined in this script using the re module to

match several token kinds (numbers, parentheses, and math operations) in the input string. The input string is subsequently tokenized into meaningful pieces using these regular expressions.

The token_type enum class has five declared token types: START, MATH_OPERATION, OPEN_PARENTHESIS, CLOSE_PARENTHESIS, and NUMBERS. Each token type is automatically assigned a unique value using the auto() function from the Enum module.
The appropriate transitions between token kinds are specified in the transitions dictionary. It states, for instance, that NUMBERS or another OPEN_PARENTHESIS may come after an OPEN_PARENTHESIS. In the same way, a MATH_OPERATION or another CLOSE_PARENTHESIS may come after a CLOSE_PARENTHESIS. For every sort of token, there are regular expressions in the data dictionary. Tokens of each class are identified by matching the input string with these regular expressions.

```
class Lexer:
    Arina Pereteatcu
    def __init__(self,equation):
        self.equation = equation

    2 usages (1 dynamic)   Arina Pereteatcu *
    def lexer(self):
        self.equation = self.equation.replace(" ", "")
        seq_parenthesis = []
        category_mapping = [token_type.START]
        failed_on = ""
        valid_tokens = []

        for symbol in self.equation:
            # Parenthesis handling
            if symbol in data[token_type.OPEN_PARENTHESIS]:
                seq_parenthesis.append(symbol)
            elif symbol in data[token_type.CLOSE_PARENTHESIS]:
                if not seq_parenthesis:
                    print(f"ERROR: Extra closing parenthesis found.")
                    print(f"Failed on symbol {failed_on}")
                    return False
                else:
                    last_open = seq_parenthesis.pop()
```

Figure2. Class Lexer

This Lexer class is designed to tokenize a given mathematical equation.
First, replace(" ", "") is used to eliminate any spaces from the equation string. To keep track of open parenthesis found during tokenization, it initializes a list called seq_parenthesis. The START token type is initialized in category_mapping, signifying the start of the tokenization procedure.

failed_on is used to record the symbols for which tokenization fails; it starts off as an empty string. The list called valid_tokens contains the valid tokens that were taken out of the equation. Every symbol in the equation is iterated over by the technique. The symbol is added to seq_parenthesis if it is an open parenthesis. The data dictionary, which has regular expressions that correspond to different token kinds, is iterated through in order to categorize the symbol. The matching category is assigned to current_category if the symbol matches any of the regular expressions. In the event that no match is discovered, an error message stating that the symbol is unclassified is returned.

```
# Transition checking
if current_category not in transations[category_mapping[-1]]:
    print(f"ERROR: Transition not allowed from '{category_mapping[-1]}' to '{current_category}'.")
    print(f"Failed on symbol {failed_on}")
    return False
```

Figure3. Transition Check

Using the transitions dictionary as a guide, it determines if moving from one token category to another is permitted. It returns an error message if the transition is not permitted.

After processing all symbols, it checks if there are any remaining open parentheses. If there are, it returns an error indicating that not all parentheses were closed.
Finally, it returns category_mapping and valid_tokens if the tokenization process completes successfully.

```
class Parser:
    Arina Pereteatcu
    def __init__(self, category_mapping, valid_tokens):
        self.category_mapping = category_mapping
        self.valid_tokens = valid_tokens

    1 usage   Arina Pereteatcu
    def parse(self):
        root = Node(self.category_mapping[0].name)
        parent = root
        for token, category in zip(self.valid_tokens, self.category_mapping[1:]):
            node = Node(token, parent=parent)
            parent = Node(category.name, parent=parent)

        for pre, _, node in RenderTree(root):
            print("%s%s" % (pre, node.name))
```

Figure4. Class Parser

This Parser class is designed to create a parse tree from the tokenized output generated by the Lexer. The “parse” method constructs a parse tree from the tokenized output. It begins by creating

a root node using the name of the first token type in category_mapping.
Then, it iterates through each token and its corresponding category in valid_tokens and category_mapping[1:] respectively.
For each token, it creates a new node with the token's value and sets its parent as the current parent node. It then updates the parent node to be a new node with the category's name.
This process effectively constructs a parse tree where each node represents either a token or a category.
After constructing the parse tree, it prints the tree using RenderTree from the anytree module. This function traverses the tree in pre-order and prints each node's name along with appropriate indentation to visualize the tree structure.

Screenshots

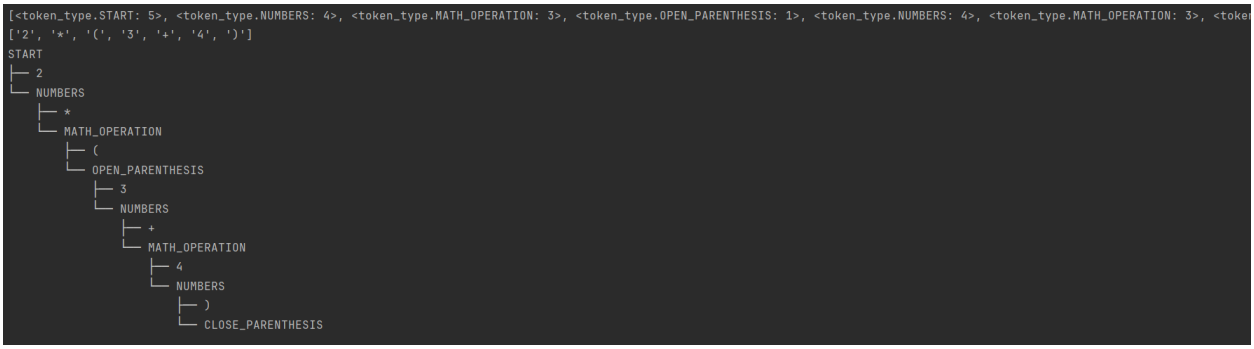


Figure5. test_equation = "2*(3+4)"

The list of token types (category_mapping) and the list of associated tokens (valid_tokens) make up the two primary components of the output. While each value in valid_tokens represents the actual tokens taken out of the equation, each element in category_mapping indicates a type of token encountered during tokenization. This tokenization is the foundation upon which the parse tree is built. The root node in the parse tree representation is the START token type. A hierarchical structure of tokens and their categories is represented by each level that follows in the tree. The parse tree visualization's indentation shows the hierarchical link between nodes and how categories and tokens are nested inside one another in accordance with the structure of the equation.

Conclusions

In this lab, we explored the compiler's syntactic analysis phase, paying particular attention to parsing. Creating a parse tree from a stream of tokens that are received from the lexical analyzer is the process of parsing. Verifying if the tokens follow the language's syntactic rules requires the parse tree. The parser ensures that the input complies with the source language's grammar by detecting and correcting grammatical errors. We looked at various parsing approaches, such as top-down and bottom-up methods.

The lexer method's lexical analysis procedure cleans the input equation, uses regular expressions to classify symbols, verifies transitions between token categories, and reports problems if any are found. This procedure makes sure that equations are tokenized correctly and gets the input ready for parsing.

The Parser` class, which houses the parsing process, creates ASTs using valid tokens that are retrieved from lexical analysis and category mappings. Tokens and mappings are iterated through by the `Parser class's `parse` method, which creates nodes in the AST for each token and its matching category. The syntactic structure of the equation is clearly visualized when the output AST is displayed in a hierarchical style using the `anytree library.