MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory work 6:
Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Bostan Victor, FAF-222

Chișinău, 2024

THEORY

Introduction to Parsing

Parsing is a fundamental process in computer science, primarily used to extract syntactical meaning from text. It is integral to the functioning of compilers and interpreters, where it involves analyzing a string of symbols based on formal grammar rules. The primary outcome of parsing is the construction of a parse tree, which outlines the syntactic structure of the input and may also embed semantic information useful for subsequent compilation stages. This capability is crucial for the software to interpret and manipulate the input code or data accurately, making parsing a cornerstone of programming language implementation and data processing.

Understanding Abstract Syntax Trees (AST)

An Abstract Syntax Tree (AST) represents the syntactic structure of input text through a hierarchy of abstraction layers. Unlike simple parse trees, ASTs focus on the high-level constructs of the text, omitting unnecessary syntactic details. Each node in the AST corresponds to a specific construct in the input, organized to reflect their syntactic and sometimes semantic relationships.

ASTs are invaluable in software development, particularly in the areas of compilation and code optimization. By abstracting the input text into a structured tree format, ASTs allow for more efficient analysis and manipulation of code. They facilitate various compiler optimizations by providing a clear framework for implementing transformation rules and conducting static code analysis efficiently.

Together, parsing and AST construction provide the tools necessary for effective software development, enabling precise control over the processing of programming languages and data formats. By transforming text into structured data, these processes support a wide range of applications, from simple code compilation to complex software engineering tasks.

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:
        a) Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
        b) 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 DESCRIPTION

This laboratory serves as a continuation of the third laboratory, as we had to implement a parser and AST generator for our lexer. So, building on the third lab I implemented these elements in the context of css code. In this project, I employed the `re` module to utilize regular expressions for tokenization in lexical analysis, effectively categorizing different segments of CSS text into tokens like selectors and properties. In fact the re library is used for the lexer and was explained in lab 3. I defined a `TokenType` enumeration using the `enum` module to systematically categorize these tokens, enhancing code clarity and maintainability. Additionally, the `graphviz` module was crucial for constructing and visualizing the Abstract Syntax Tree (AST), providing a graphical depiction of node and edge relationships in the CSS structure.

So, as I already mentiond I used the lexer developed in lab 3. This lexer is able to understand and tokenize css code:

```
21  # Lexer function to tokenize the input CSS content
22  def lexer(css):
23      tokens = []
24      while css:
25          css = css.strip()
26          match_found = False
27          for token_type, token_regex in TOKENS:
28              regex = re.compile(token_regex)
29              match = regex.match(css)
30              if match:
31                  value = match.group(0).strip()
32                  if token_type != TokenType.WHITESPACE and token_type != TokenType.COMMENT:
33                      tokens.append((token_type, value))
34                  css = css[match.end():]
35                  match_found = True
36                  break
37          if not match_found:
38              raise SyntaxError(f'Unknown CSS: {css}')
39      return tokens
```

Now the first step of this lab was to create the enum TokenType in order to categorize each type of token in my css lexer. So, here it is:

```
# Define the TokenType using an enumeration for clarity and robustness
class TokenType(enum.Enum):
    WHITESPACE = 1
    COMMENT = 2
    AT_RULE = 3
    SELECTOR = 4
    BRACE_OPEN = 5
    BRACE_CLOSE = 6
    PROPERTY = 7
    FUNCTION = 8
    FUNCTION_ARGS = 9
    VALUE = 10
    COLON = 11
    SEMICOLON = 12
    COMMA = 13
```

Following the categorization of CSS tokens using the TokenType enumeration, the next crucial step in our parsing process involves constructing the Abstract Syntax Tree (AST). For simplicity and flexibility, I implemented a single node class, ASTNode, which represents all elements within the CSS structure, whether they are selectors, properties, or values:

```
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 []
```

This class encapsulates all necessary attributes for any node in the AST: type (indicating the kind of token), value (the specific textual content of the token), and children (a list of sub-nodes that fall under this node in the hierarchy). Now that I defined the nodes it's time to build the parse function. The parse function is designed to read through the tokens and systematically build the AST by nesting nodes according to their syntactic relationships:

```python
# Parser function to build AST from tokens
def parse(tokens):
    root = ASTNode(TokenType.SELECTOR, value="ROOT")  # Use enum for ROOT
    current_node = root

    i = 0
    while i < len(tokens):
        token_type, value = tokens[i]
        if token_type == TokenType.SELECTOR:
            current_node = ASTNode(token_type, value=value)
            root.children.append(current_node)
        elif token_type == TokenType.PROPERTY:
            property_node = ASTNode(token_type, value=value)
            current_node.children.append(property_node)
            # Expecting a VALUE token next
            if i+2 < len(tokens) and tokens[i+2][0] == TokenType.VALUE:
                value_node = ASTNode(tokens[i+2][0], value=tokens[i+2][1])
                property_node.children.append(value_node)
                i += 2  # Move past the VALUE token
        i += 1

    return root
```

The function starts by establishing a ROOT node which acts as the initial anchor point for the AST. This node, although not corresponding to any actual CSS selector, serves as a conceptual base for building the tree structure.

As the function iterates over the array of tokens, it utilizes a systematic approach to decide the placement of each token within the AST based on its type. When a SELECTOR token is encountered, a new node is created and added directly under the root. This node will then act as the current point of focus for any subsequent properties or values until a new selector is identified.

For tokens identified as PROPERTY, a new node is also created and attached as a child to the current selector node, reflecting their natural hierarchical relationship in CSS. If the function detects a VALUE token immediately following a property—typically within the next two tokens—it further creates a node for this value and nests it under the property node. This structure mimics the standard CSS syntax where properties are directly followed by their values.

The final phase of the AST construction involves visualizing this structure using Graphviz, which helps in debugging and better understanding the hierarchical organization:

```python
def add_nodes_edges(tree, graph=None):
    if graph is None:
        graph = Digraph()
        # Include both the type and value in the label for the root node
        graph.node(name=str(id(tree)), label=f'{tree.type.name}({tree.value})')

    for child in tree.children:
        # Include both the type and value in the label for each child node
        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
```

Initially, the function checks if a graph object already exists; if not, it creates a new Graphviz diagram (Digraph). It then proceeds to add a graphical node for the root of the AST. This node is labeled with both the type and the value of the root, providing immediate visual feedback on the root's characteristics. This label is important as it combines the node type (e.g., SELECTOR, PROPERTY) with its actual content from the CSS, making it easier to understand at a glance.

As the function iterates through each child of the current node, it recursively applies the same process: it adds a node for the child to the graph and creates an edge from the current node (parent) to the child node. This effectively draws the connections that exist within the AST, illustrating the parent-child relationships that form the hierarchical structure of the CSS document. Each child node is similarly labeled with its type and value, ensuring that each element of the CSS is represented clearly in the visualization.

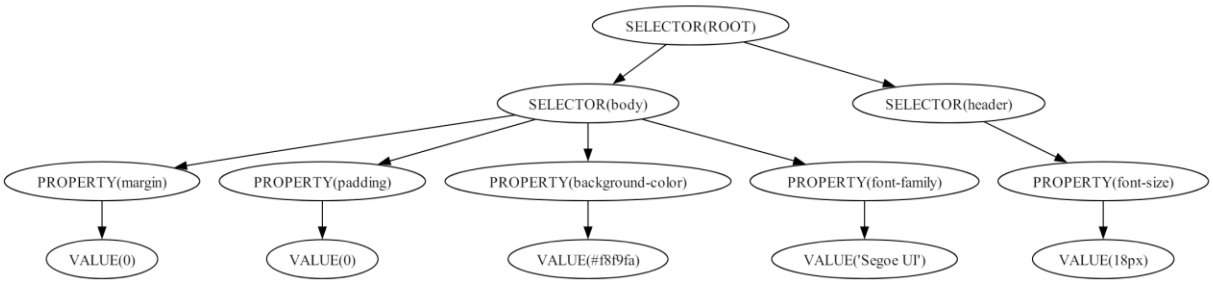
Results:

So, to demonstrate how everything works I used this sample CSS code:

```css
/* Main body styles */
body {
    margin: 0;
    padding: 0;
    background-color: #f8f9fa;
    font-family: 'Segoe UI';
}

/* Media query for mobile devices */
@media (max-width: 600px) {
    header {
        font-size: 18px;
    }
}
```

Here is the generated Abstract Syntax Tree:



CONCLUSION

The laboratory work conducted has significantly deepened my understanding of parsing and the use of Abstract Syntax Trees (AST), crucial tools for the analysis of structured text. By implementing a parser and using Graphviz for visualization, I effectively transformed CSS text into a hierarchical AST, which accurately reflects the syntactical organization of the input. This experience has not only reinforced my grasp of parsing concepts but also enhanced my practical skills in building and visualizing complex data structures. Through this process, I learned to systematically handle and categorize textual data, skills that are essential for my future projects in software development and computational linguistics. Overall, the lab work was an invaluable practice in applying theoretical knowledge to real-world applications, preparing me for more advanced challenges in the field.