

Topic: Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

Parsers and Abstract Syntax Trees (ASTs) are critical components in the compilation and interpretation of programming languages, as well as in various applications of structured text processing. Here's an overview of their basic theory and applications:

1. **Parsing:** The primary function of a parser is to analyze a sequence of tokens (generated by a lexer) to determine its grammatical structure with respect to a given formal grammar. This involves recognizing and enforcing the syntactic structure of the language.
2. **Context-Free Grammars (CFG):** Parsers commonly rely on context-free grammars to define the rules for syntactic structure. CFGs consist of a set of production rules that describe how symbols of the language can be combined to form correctly structured phrases or sentences.
3. **Parsing Techniques:** There are two main types of parsing techniques:
 - **Top-Down Parsing:** Begins analysis from the highest level (start symbol) and moves toward the leaves, using the production rules to expand symbols. Recursive descent parsing is a common example of a top-down approach.
 - **Bottom-Up Parsing:** Starts with the input tokens and attempts to construct the parse tree by working up towards the start symbol. Shift-reduce parsing, including the popular LR and LL parsers, are examples of this approach.
4. **Parse Tree and Abstract Syntax Tree (AST):** A parse tree is a tree that represents the syntactic structure of the input token sequence as described by the grammar. An Abstract Syntax Tree, or AST, is a more compact version of a parse tree,

constructed by abstracting away certain details like punctuation and grouping tokens into expressive constructs.

5. **Semantics Handling:** Once the AST is built, it can be used to attach semantic meaning to each construct, facilitating further processing steps such as type checking, semantic analysis, and code generation in compilers.
6. **Error Handling:** Parsers are also responsible for detecting syntax errors in the input sequence. Effective error handling in parsers is crucial for providing clear, useful feedback about syntax errors to the user.
7. **Output:** The output of the parsing phase is typically an AST which provides a structured, easy-to-process representation of the input code, suitable for further stages like semantic analysis and code generation.

Applications:

Programming Languages: Parsers and ASTs form the backbone of compilers and interpreters, transforming code into structured formats that machines can execute or interpret further.

Query Languages: In databases, parsers are used to interpret and execute queries written in SQL or other query languages, translating user commands into operations that the database can perform.

Document Processing: Parsers are used in the processing of complex document formats that have hierarchical structures, such as XML and HTML, ensuring documents adhere to their DTDs or schemas.

Natural Language Processing: Parsers help in understanding and structuring natural language with grammars that define the composition of sentences, which is crucial for applications such as speech recognition and machine translation.


Overall, the theory behind parsers and ASTs is deeply embedded in computer science, particularly in the fields of compiler construction, formal language theory, and the broader domain of language processing technologies. These tools are essential for translating and structuring raw text into meaningful constructs that facilitate a wide range of computing applications.

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

Implementation description:

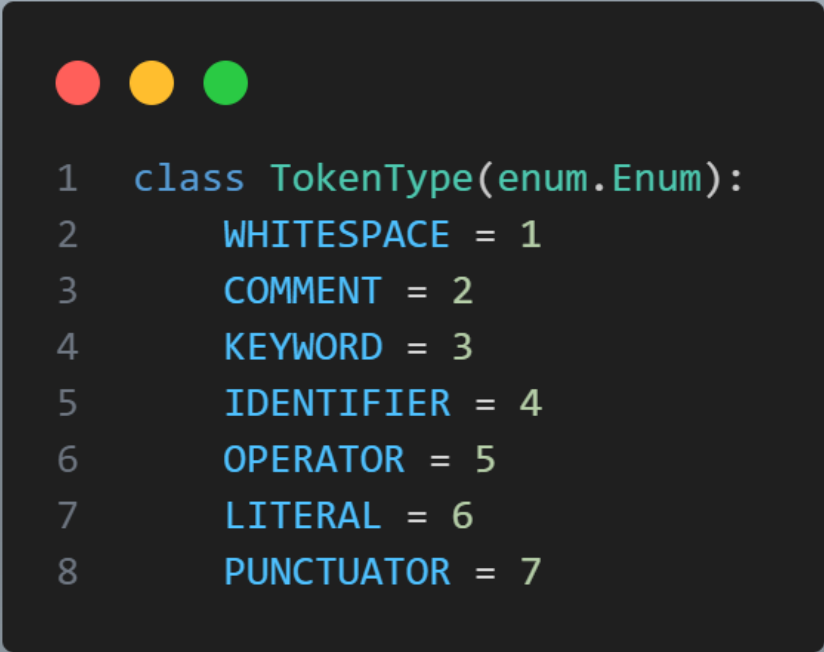
Imports and Setup:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains three lines of Python code:

```
1 import re
2 import enum
3 from graphviz import Digraph
```

- **Modules:** The script imports `re` for regular expression operations, `enum` for defining enumeration classes, and `graphviz` for visualizing the AST.
- **TokenType Enumeration:** Defines `TokenType` as an enum to categorize various token types encountered in C++, such as keywords, identifiers, literals, and operators.

Token Type Definition:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains eight lines of Python code defining the TokenType enumeration:

```
1 class TokenType(enum.Enum):
2     WHITESPACE = 1
3     COMMENT = 2
4     KEYWORD = 3
5     IDENTIFIER = 4
6     OPERATOR = 5
7     LITERAL = 6
8     PUNCTUATOR = 7
```

This portion of the code defines an enumeration, `TokenType`, which categorizes the types of tokens that can be recognized in the input text. The enum includes various token types that are commonly found in programming languages:

- **WHITESPACE**: Represents spaces, tabs, and other whitespace characters.
- **COMMENT**: Represents comments, which can be either single-line (starting with `//`) or multi-line (enclosed between `/*` and `*/`).
- **KEYWORD**: Represents reserved words in a language that have special meaning, such as `if`, `else`, `while`, etc.
- **IDENTIFIER**: Represents names given to entities like variables and functions.
- **OPERATOR**: Represents symbols that denote operations, like `+`, `-`, `=`, etc.
- **LITERAL**: Represents fixed values in code, like numbers.
- **PUNCTUATOR**: Represents punctuation characters that are used to structure the code, like commas, semicolons, and braces.

Tokens Array:

```
1  TOKENS = [  
2      (TokenType.WHITESPACE, r'\s+'),  
3      (TokenType.COMMENT, r'//.*|/\*[\^]*\*+([^\*][\^]*\*+)*\/'),  
4      (TokenType.KEYWORD, r'\b(int|char|float|double|return|if|else|while|for)\b'),  
5      (TokenType.IDENTIFIER, r'\b[_a-zA-Z][_a-zA-Z0-9]*\b'),  
6      (TokenType.LITERAL, r'\b\d+(\.\d+)?\b'),  
7      (TokenType.OPERATOR, r'[+|-|*|/|=|<|>|!&|^|+|)'),  
8      (TokenType.PUNCTUATOR, r'[\{\}\(\);,]'),  
9  ]
```

This array consists of tuples where each tuple associates a `TokenType` with a regular expression that defines how to recognize tokens of that type:

- **WHITESPACE:** Captured by `\s+` which matches one or more whitespace characters.
- **COMMENT:** Captured by a complex regular expression that matches both single-line and multi-line comments.
- **KEYWORD:** Captured by `\b(...)\b` where ... is a pipe-separated list of specific keywords. The `\b` ensures that these are matched as whole words.
- **IDENTIFIER:** Matches sequences that start with a letter or underscore, followed by any number of letters, digits, or underscores.
- **LITERAL:** Matches integers and floating-point numbers.
- **OPERATOR:** Matches one or more characters from a set of arithmetic and logical operators.
- **PUNCTUATOR:** Matches single characters used for punctuation in the code.

Lexer Function:

```

1  def lexer(cpp):
2      tokens = []
3      while cpp:
4          cpp = cpp.lstrip()
5          match_found = False
6          for token_type, token_regex in TOKENS:
7              regex = re.compile(token_regex)
8              match = regex.match(cpp)
9              if match:
10                 value = match.group(0).strip()
11                 if token_type != TokenType.WHITESPACE:
12                     tokens.append((token_type, value))
13                 cpp = cpp[match.end():]
14                 match_found = True
15                 break
16             if not match_found:
17                 raise SyntaxError(f'Unknown C++ syntax: {cpp}')
18     return tokens

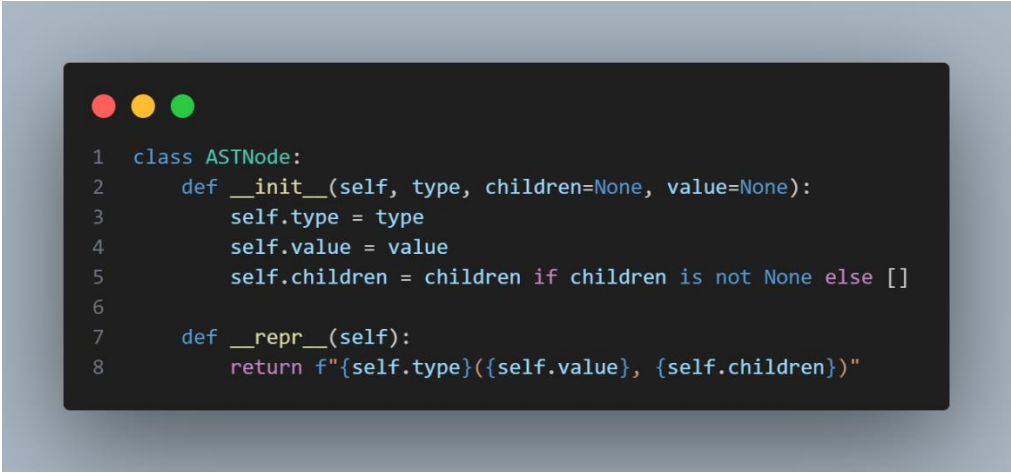
```

The lexer function processes a string of input code to extract tokens

based on the patterns defined in the TOKENS array:

- Initialization: Starts with an empty list of tokens.
- Processing Loop: Continues as long as there is code left to process. It removes leading whitespace and then iterates through the token patterns to find matches at the start of the remaining code.
- Token Recognition: For each pattern, if a match is found, the matched text is extracted, and, unless it's whitespace, it's added to the token list. The code is then updated to remove the processed portion.
- Error Handling: If no pattern matches the beginning of the code, a `SyntaxError` is raised indicating unrecognized syntax.

AST Node Class:

A code editor window with a dark background and light-colored text. It shows the definition of the `ASTNode` class. The code is as follows:

```
1 class ASTNode:
2     def __init__(self, type, children=None, value=None):
3         self.type = type
4         self.value = value
5         self.children = children if children is not None else []
6
7     def __repr__(self):
8         return f"{self.type}({self.value}, {self.children})"
```

Defines a node in the abstract syntax tree (AST):

- Initialization: Each node stores a type (such as "Program", "Declaration", etc.), a value (optional), and a list of children nodes.
- Representation: The `__repr__` method provides a string representation of the node for debugging and display purposes, showing the node type, value, and children.

Parser Function:

```
1 def parse(tokens):
2     root = ASTNode("Program")
3     current_node = root
4     stack = []
5
6     i = 0
7     while i < len(tokens):
8         token_type, value = tokens[i]
9         if token_type == TokenType.KEYWORD and value in ["int", "char", "float", "double"]:
10             if i + 1 < len(tokens) and tokens[i + 1][0] == TokenType.IDENTIFIER:
11                 declaration_node = ASTNode("Declaration", value=value + " " + tokens[i + 1][1])
12                 current_node.children.append(declaration_node)
13                 i += 1
14             elif value == '{':
15                 stack.append(current_node)
16                 current_node = ASTNode("Block")
17                 stack[-1].children.append(current_node)
18             elif value == '}':
19                 current_node = stack.pop()
20             elif value == '=':
21                 if i + 1 < len(tokens) and tokens[i + 1][0] == TokenType.LITERAL:
22                     assignment_node = ASTNode("Assignment", value=tokens[i - 1][1] + " = " + tokens[i + 1][1])
23                     current_node.children.append(assignment_node)
24                     i += 1
25                 i += 1
26
27     return root
```

Function Definition and Initial Setup

- Line 1: The parse function is defined with one parameter, tokens, which is a list of token tuples where each tuple contains a TokenType and its corresponding value.
- Line 2: An ASTNode named root with the type "Program" is created as the root of the AST.
- Line 3: current_node is initialized to root. This variable will keep track of the current position in the AST as nodes are added.
- Line 4: stack is an empty list that will be used to manage nested structures like blocks of code, essentially remembering where to return after closing a block.

Parsing Loop

- Line 6-27: The function uses a while loop to iterate through each token in the tokens list, processing each based on its type and value.

Token Handling

- Lines 8-9: Each token's type and value are extracted. The parsing behavior depends on these values.
- Lines 10-13: If the token is a KEYWORD and one of ["int", "char", "float", "double"] and is followed by an IDENTIFIER, a new ASTNode of type "Declaration" is created. This node takes the keyword and identifier as its value (e.g., "int x"), which signifies a variable declaration. This node is then added as a child to the current_node.
- Line 14: If a { is encountered, it indicates the start of a new block. The current node is pushed onto the stack, and a new ASTNode of type "Block" is created and set as the new current_node.
- Line 18: If a } is encountered, it signifies the end of the current block. The current_node is set back to the node on top of the stack, effectively returning to the previous level of the AST.
- Lines 20-24: If an = is encountered, and it's followed by a LITERAL, an "Assignment" node is created with the variable name and the literal value as its content (e.g., "x = 10"). This represents an assignment operation in the code. This node is added as a child to the current_node.
- Line 25: The index i is incremented to continue to the next token.
- Line 27: After all tokens are processed, the root of the fully constructed AST is returned.

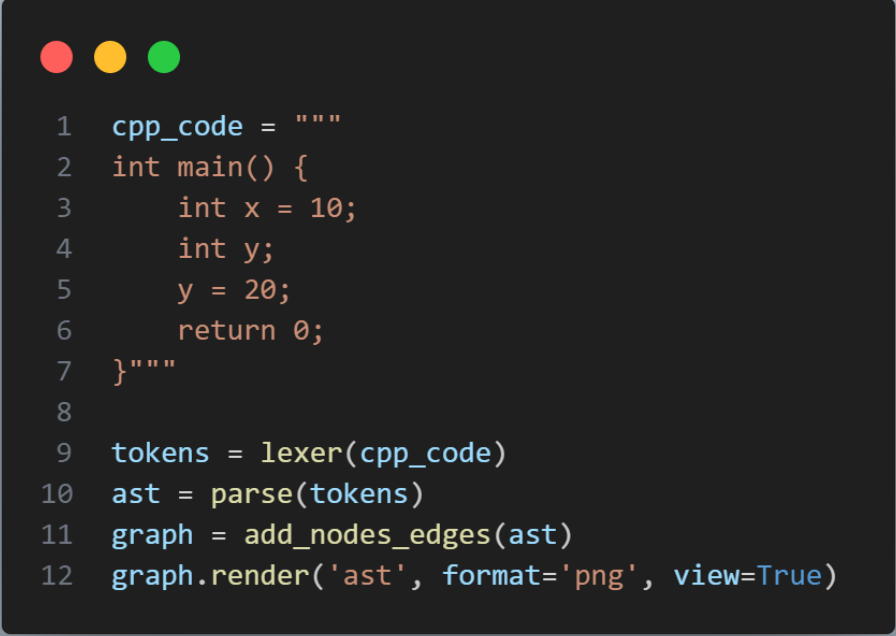
This parsing approach builds an AST that represents the hierarchical structure of a simple program, handling variable declarations, blocks, and assignments. Each node in the AST corresponds to a specific construct in the parsed code.

Visualization Function:

```
1 def add_nodes_edges(tree, graph=None):
2     if graph is None:
3         graph = Digraph()
4         graph.node(name=str(id(tree)), label=f'{tree.type}({tree.value})')
5
6     for child in tree.children:
7         graph.node(name=str(id(child)), label=f'{child.type}({child.value})')
8         graph.edge(str(id(tree)), str(id(child)))
9         add_nodes_edges(child, graph)
10
11     return graph
```

This function adds nodes and edges to a Graphviz graph based on the structure of an Abstract Syntax Tree (AST). It takes an AST node and an optional graph object as parameters. If no graph is provided, it creates a new one. It labels each node with the type and value of the AST node and connects nodes to represent parent-child relationships. The function recursively processes each child of the AST node, adding nodes and connecting them with edges until the entire tree is represented in the graph.

Example Code Block for Creating and Visualizing an AST:



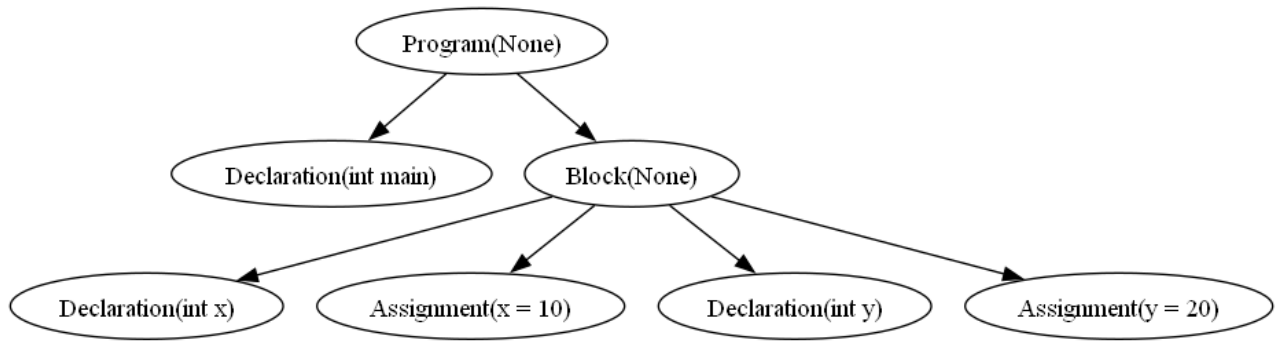
```
1  cpp_code = """
2  int main() {
3      int x = 10;
4      int y;
5      y = 20;
6      return 0;
7  }"""
8
9  tokens = lexer(cpp_code)
10 ast = parse(tokens)
11 graph = add_nodes_edges(ast)
12 graph.render('ast', format='png', view=True)
```

This block demonstrates how to analyze a simple C++ program and visualize its structure as an AST:

- A string `cpp_code` contains a C++ program that includes variable declarations, assignments, and a return statement.
- The `lexer` function is called to tokenize the C++ code, breaking it down into recognizable symbols.
- The `parse` function takes these tokens and constructs an AST, organizing the tokens into a hierarchical tree structure that reflects the code's syntax.
- The `add_nodes_edges` function is then used to create a visual representation of this AST using Graphviz, which visually maps out the tree structure of the code.
- Finally, the graph is rendered as a PNG image and displayed, showing a graphical view of the AST.
- This process transforms textual code into a graphical diagram of its structural elements, making it easier to understand the relationships and hierarchy within the program.

Conclusions/Screenshots/Results:

This is the output of the parse and visualization functions for the C++ Sample Code provided in the code as input:



Throughout this laboratory work focused on constructing a parser, I engaged deeply with the principles of syntactic analysis crucial for building compilers. The experience of crafting a parser to convert sequences of tokens from a lexer into an organized Abstract Syntax Tree (AST) was particularly enlightening, illustrating the key role parsers play in understanding and processing programming languages.

Developing a parser as demonstrated in the provided code involved not only implementing a lexer to handle preliminary tokenization based on regular expressions but also designing the logic to form a structured AST. This allowed practical application of theoretical knowledge such as grammars, tokens, and node types in a hands-on environment. Implementing features to handle different types of tokens like keywords, identifiers, literals, and operators was especially challenging and informative.

By creating nodes for each significant syntax element (declarations, assignments, and blocks) and managing their hierarchical relationships within the AST, I gained insights into the complexity of code structure in higher-level programming languages. This

laboratory work also highlighted the importance of meticulous design and testing in parser construction to ensure accurate interpretation of source code.

In conclusion, this laboratory work has been instrumental in enhancing my understanding of parsers and their critical function in the compilation process. It has equipped me with valuable practical skills in compiler construction that will be beneficial for my future projects and endeavors in computer science.

References:

<https://introcs.cs.princeton.edu/java/51language/>

<https://www.geeksforgeeks.org/introduction-of-finite-automata/>