Laboratory Work Nr.6 Abstract Syntax Tree Parsing

Cretu Cristian

Course: Formal Languages & Finite Automata

Group: FAF-223

1 Theory

An abstract syntax tree (AST) is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text. It is sometimes called just a syntax tree. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches. This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis. Abstract syntax trees are also used in program analysis and program transformation systems. Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

2 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:

(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.

3 Implementation Description

3.1 `TokenType` Class

```
import enum

class TokenType(enum.Enum):
    DOCUMENT_OPEN = 1
    DOCUMENT_CLOSE = 2
    TITLE_OPEN = 3
    TITLE_CLOSE = 4
    CHAPTER_OPEN = 5
    CHAPTER_CLOSE = 6
    SUBCHAPTER_OPEN = 7
    SUBCHAPTER_CLOSE = 8
    CONTENT = 9

    def __eq__(self, value: object) -> bool:
        return super().__eq__(value)
```

3.2 Lexer

```
def lexer(markdown):
    tokens = []
    while markdown:
        markdown = markdown.strip()
        match_found = False
        for token_type, token_regex in TOKENS:
            regex = re.compile(token_regex)
            match = regex.match(markdown)
            if match:
                value = match.group(0).strip()
                tokens.append((token_type, value))
                markdown = markdown[match.end():]
                match_found = True
                break
        if not match_found:
            raise SyntaxError(f'Unknown markdown: {markdown}')
    return tokens
```

3.3 Tokens

```
TOKENS = [
    (TokenType.DOCUMENT_OPEN, r'\[document\]'),
    (TokenType.DOCUMENT_CLOSE, r'\[/document\]'),
    (TokenType.TITLE_OPEN, r'\[title\]'),
    (TokenType.TITLE_CLOSE, r'\[/title\]'),
    (TokenType.CHAPTER_OPEN, r'\[chapter\]'),
    (TokenType.CHAPTER_CLOSE, r'\[/chapter\]'),
    (TokenType.SUBCHAPTER_OPEN, r'\[subchapter\]'),
    (TokenType.SUBCHAPTER_CLOSE, r'\[/subchapter\]'),
    (TokenType.CONTENT, r'[^\[\]]*'),  % Match any content not containing '[' or ']'
]
```

3.4 Abstract Syntax Tree Node Class

```
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 []

    def __repr__(self):
        type_name = self.type.name if isinstance(self.type, enum.Enum) else self.type
        return f"{type_name}({self.value}, {self.children})"
```

3.5 Parser Class

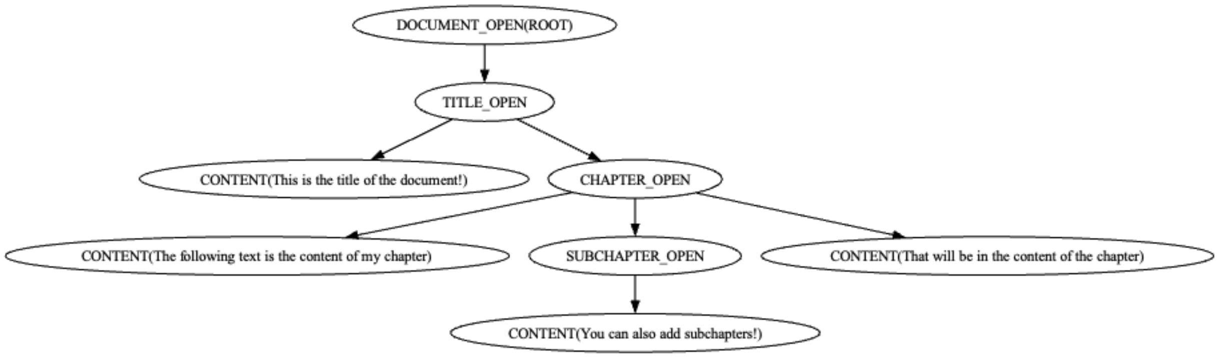
```
class Parser:
    def __init__(self):
        self.root = None
        self.current_node = None
        self.stack = []

    def parse(self, tokens):
        self.root = ASTNode(TokenType.DOCUMENT_OPEN, value="ROOT")
        self.current_node = self.root
        self.stack = [self.root]

        for token_type, value in tokens:
            self.handle_token(token_type, value)

        return self.root

    def handle_token(self, token_type, value):
        if token_type == TokenType.CHAPTER_OPEN:
            self.handle_chapter_open()
        elif token_type == TokenType.SUBCHAPTER_OPEN:
            self.handle_subchapter_open()
        elif token_type == TokenType.TITLE_OPEN:
            self.handle_title_open()
        elif token_type == TokenType.CONTENT:
            self.handle_content(value)
        elif token_type == TokenType.CHAPTER_CLOSE:
            self.handle_chapter_close()
        elif token_type == TokenType.SUBCHAPTER_CLOSE:
            self.handle_subchapter_close()
        elif token_type == TokenType.DOCUMENT_CLOSE:
            self.handle_document_close()

    def handle_chapter_open(self):
        chapter_node = ASTNode(TokenType.CHAPTER_OPEN)
        self.current_node.children.append(chapter_node)
        self.stack.append(chapter_node)
        self.current_node = chapter_node
```

4 Execution and Result

To test and present this lab, I have created a file `example.flexksx` with following lines. It has a very simple syntax.

```
[document]
[title]
    This is the title of the document!
[/title]
[chapter]
    The following text is the content of my chapter
        [subchapter]
            You can also add subchapters!
        [/subchapter]
    That will be in the content of the chapter
[/chapter]
[/document]
```

The processing of such code will result in the following tree:


DOCUMENT_OPEN(ROOT)
TITLE_OPEN
CONTENT(This is the title of the document!)
CHAPTER_OPEN
CONTENT(The following text is the content of my chapter)
SUBCHAPTER_OPEN
CONTENT(That will be in the content of the chapter)
CONTENT(You can also add subchapters!)

Where we can clearly see a hierarchy of the document structure defined in the custom language. We observe that the document opens, then it has a title, then inside the title there is a chapter defined, which has its content, a subchapter with its own distinct content, and the content that follows after the subchapter. This indicates that the execution was correct and we achieved the purpose of parsing and building an AST Tree.

5 Conclusion

In conclusion, this laboratory work provided an in-depth exploration of parsing techniques and abstract syntax trees (ASTs), fundamental concepts in formal language processing. Through the implementation of a lexer and parser for a custom markup language, the lab aimed to demonstrate practical applications of parsing methodologies. By defining a `TokenType` enumeration and employing regular expressions for token identification, the lexer efficiently transformed markdown text into a sequence of tokens. Subsequently, the `Parser` class orchestrated the construction of an AST from the token stream, utilizing hierarchical document structures defined in the input markdown. The modular design of the `Parser` class facilitated extensibility and maintainability, while the `ASTNode` class provided a flexible representation of nodes within the AST. Execution of the lab's example code successfully produced an AST reflecting the hierarchical structure of the input document, validating the efficacy of the parsing process. Overall, the lab underscored the importance of parsing techniques and ASTs in formal language processing, laying a solid foundation for further exploration and application in compiler construction and program analysis domains.