

laboratory_6.md


Laboratory Work 6

 **Course: Formal Languages & Finite Automata**

 **Author: Felicia Novac**




Theory

 Parsing is the process of analyzing a sequence of tokens to determine its grammatical structure according to a set of predefined rules or a grammar. It is used to convert a stream of text, code, or data into a structured format, like a parse tree or an abstract syntax tree, which represents the relationships and hierarchy within the input. The parsing is divided into two types, which are as follows:

- Top-down parsing - building the parse tree from the root node to the leaf node.
- Bottom-up parsing - building the parse tree from the leaf node to the root node.

Objectives:

- 
- Get familiar with parsing, what it is and how it can be programmed.
 - Get familiar with the concept of AST.
 - Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
 - 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

- 🔗 It is important to mention that I haven't had a `TokenType` enum within my project, so first of all I slightly reformatted the `Tokenizer` and `Token` classes to introduce it. After the adjustments I could continue with implementing the parser, as I've initially used REGEX within laboratory 3, so no adjustments needed here.

Data Structure

- 🔗 The `ASTNode` class defines the nodes in an Abstract Syntax Tree (AST). It has three key attributes: a `Token` that contains the node's value, a list of children representing child nodes, and a parent reference pointing to the node's parent.

It contains helper methods as `Getter s` and `Setter s`, as well as `addChild(ASTNode child)` which sets the child's parent reference to the current node and appends it to the children list, maintaining the tree structure.

Parser

- 🔗 The parser, or `ASTBuilder` class is used to create an Abstract Syntax Tree (AST) by parsing a list of tokens (passed by the `Lexer`) with the top-down parsing type.

It contains a method called `ASTNode buildParseTree(List<Token> tokens)` that starts with verifying the input list to ensure it has tokens, and has some if-statements to ensure the correct command start, which is `imp --img="path"`. It also initializes a stack to manage the tree-building process. The root node is created from the first token, and subsequent nodes are added based on their type as the token list is iterated over.

Each new command token is pushed onto the stack, ensuring correct parent-child relationships. For instance, the `--img="folder\file\path"` will be considered as the child of the `imp` `Token`, and the next command will be then considered as its respective children. Also, When encountering a `PIPE_LINE`, the stack pops to transition to a previous context, reflecting a command chain or a pipeline operation.

Additionally, I've implemented a `CommandStructure` class, where a static `Map` is defined. This map holds the commands' structure of the commands containing parameters, identifying them as required.

Tokens are processed using a switch statement and connected using the `addChild` method from the `ASTNode` class. If a new command or operation is encountered, it is added as a child to the current stack's top node.

Conclusions / Screenshots / Results

In order to better visualize the hierarchical representation of the parsing result (AST), I've implemented two helper methods `printParseTree` and `printNode` for displaying the tree. This map is used when parsing Tokens identified as `COMMAND` .

So, let's analyze a few command and see the results. Note that the commands are firstly passed to the Tokenizing phase and only then received by parser.

	Examples without Pipelines	Examples with Pipelines
Valid Input Prompts	<pre>Enter the input string: imp --img="image.png" rotate --deg=90 AST: imp --img = image.png rotate --deg = 90</pre>	<pre>Enter the input string: imp --img="image.png" crop --x=100 --y=100 --w=200 --h=200 --> convert --format=jpg --> rotate --deg=90 AST: imp --img = image.png crop --x = 100 --y = 100 --w = 200 --h = 200 convert --format = jpg rotate --deg = 90</pre>
Invalid Input Prompts	<pre>Enter the input string: imp crop --x=100 --y=100 --w=200 --h=200 Exception in Thread "Main": java.lang.IllegalArgumentException: Second token must be an image identifier at ASTBuilder.buildParseTree(ASTBuilder.java:15) at Main.main(Main.java:15)</pre>	<pre>Enter the input string: imp --img="image.png" --> rotate --deg=90 Exception in Thread "Main": java.lang.IllegalArgumentException: Pipe line without a command at ASTBuilder.buildParseTree(ASTBuilder.java:57) at Main.main(Main.java:15)</pre>

Table 1. Parser in Action

Observe that I've implemented a slight error handling, with an accent on the commands' structure as the tokens' corectness was implemented within the Lexer.

In conclusion, I've managed to get the essence of Parsing and Abstract Syntax Tree (AST) concepts. I've also successfully (in my modest opinion) implemented them within code, and did my best to handle as many errors as possible. It was fun to practice with AST Data Structure, because I wanted to make it pretty and make it an actual data structure instead of just print formatting.

P.S. It was a great course and thanks to the laboratory works, I actually managed to learn something useful and practice the skills.

References

[1] **Parsing** - Accessed April 27, 2024. <https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/>.

[2] **AST** - Accessed April 27, 2024. https://en.wikipedia.org/wiki/Abstract_syntax_tree.