Topic: Chomsky Normal Form

Course: Formal Languages & Finite Automata

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:
    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.

Laboratory work:

Theoretical notes:

One of the programs that makes up the compiler is called a parser, and parsing is a step in the compilation process. Parsing takes place in the compilation's analysis phase.

Code that has been taken from the preprocessor is parsed, divided into manageable chunks, and examined so that it can be understood by other programs. The parser accomplishes this by assembling the input elements into a data structure.
The parser consists of three components, each of which handles a different stage of the parsing process. The three stages are:

Stage 1: Lexical analysis

A lexical analyzer -- or scanner -- takes code from the preprocessor and breaks it into smaller pieces. It groups the input code into sequences of characters called lexemes, each of which corresponds to a token. Lexical analyzers also remove white space characters, comments and errors from the input.

Stage 2: Syntactic analysis

This stage of parsing checks the syntactical structure of the input, using a data structure called a parse tree or derivation tree. A syntax analyzer uses tokens to construct a parse tree that combines the predefined grammar of the programming language with the tokens of the input string.

Stage 3: Semantic analysis

Semantic analysis verifies the parse tree against a symbol table and determines whether it is semantically consistent. This process is also known as context sensitive analysis. It includes data type checking, label checking and flow control checking.

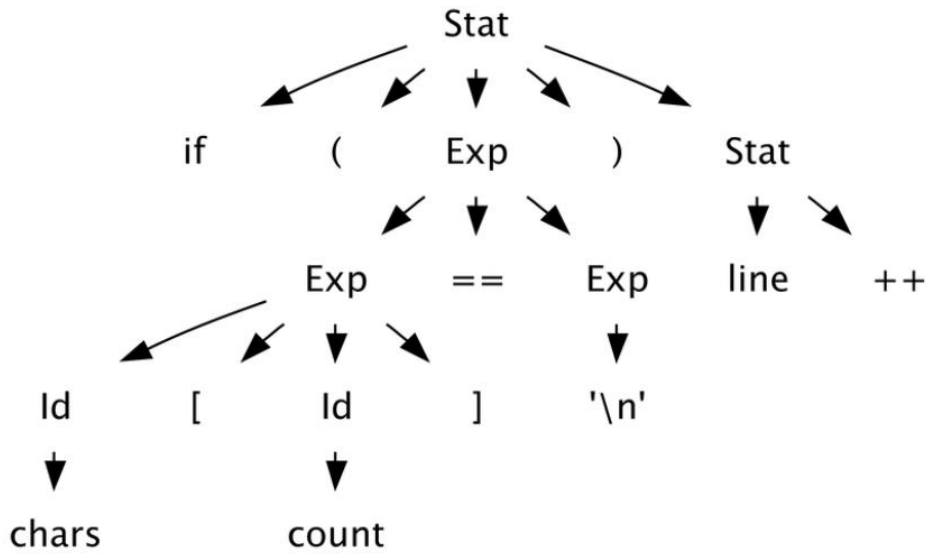
When a software language is created, its creators must specify a set of rules. These rules provide the grammar needed to construct valid statements in the language. Parsers are used when there is a need to represent input data from source code abstractly as a data structure so that it can be checked for the correct syntax. Coding languages and other technologies use parsing of some type for this purpose.

What is a Parse Tree?

A parse tree is generated by the parser, which is a component of the compiler that processes the source code and checks it for syntactic correctness. The parse tree is then used by other components of the compiler, such as the code generator, to generate machine code or intermediate code that can be executed by the target machine.

Parse trees can be represented in different ways, such as a tree structure with nodes representing the different elements in the source code and edges representing the relationships between them, or as a graph with nodes and edges representing the same information. Parse trees are typically used as an intermediate representation in the compilation process, and are not usually intended to be read by humans.

Here is how it looks for a Java statement:



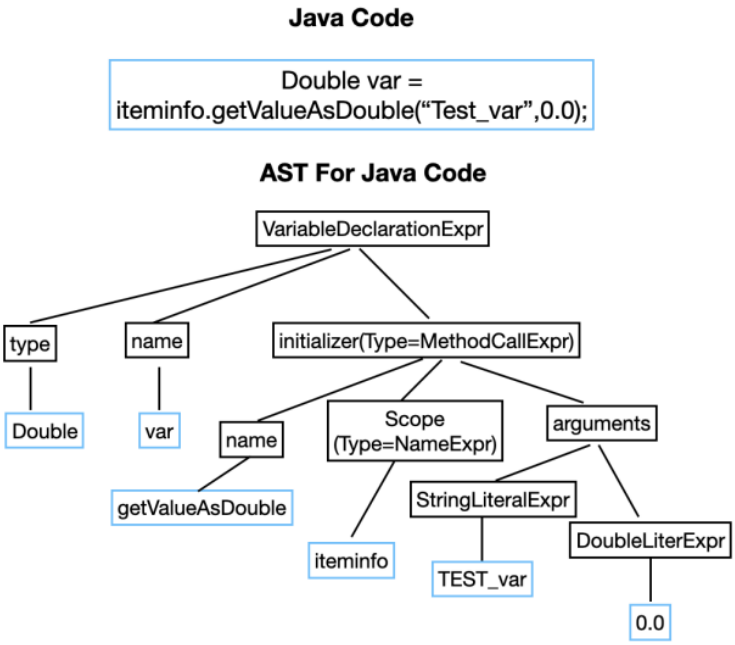
What is an Abstract Syntax Tree?

An **Abstract Syntax Tree (AST)** abstracts away certain details and retains just enough information to help the compiler understand the structure of the code. Therefore, an AST is a **tree** data structure that best represents the **syntactic** structure of the **source code**.

In general, the AST **models** the relationship between tokens in the source code as a *tree* comprising of *nodes* and the nodes containing *children*. And each node contains information about the type of token, and related data.

For example, if your node represents a function call, the arguments and the return values will be the associated data.

Here is how an AST for a java statement looks like:



Implementation

The following implementation of the parser is here:

In the constructor of the Parser class, the provided list of tokens is assigned to the tokens field. This allows the parser to work with the tokens throughout its lifecycle. The currentTokenIndex is initialized to 0, indicating that the parser will start parsing from the first token in the list.

```
public Parser(List<Lab3.AgainLab3.Token> tokens) {
        this.tokens = tokens;
        this.currentTokenIndex = 0;
    }
```

The parse() method serves as the entry point for the parsing process. It attempts to parse the entire program by calling the parseProgram() method. If parsing is successful, a message

indicating completion is printed. If an exception occurs during parsing, we'll get an error message.

```
public void parse() {
        try {
            // Parse the program on Adder
            parseProgram();
        } catch (Lab3.AgainLab3.ParseException e) {
            System.out.println("Parsing error: " + e.getMessage());
//print error message if parsing fails
        }
    }
```

The parseProgram() method is responsible for parsing the entire program. It iterates through the list of tokens and calls the parseStatement() method for each statement in the program. This method ensures that all statements in the program are parsed.

```
private void parseProgram() throws Lab3.AgainLab3.ParseException {
        while (currentTokenIndex < tokens.size()) {
            parseStatement(); // Call the method to parse each statement
        }
    }
```

Within the parseStatement() method, individual statements are parsed. At this point, you would typically handle different types of statements based on the language grammar. For simplicity, each statement is currently printed. The consume() method is called to retrieve the current token being processed, and its content is printed as the parsed statement.

```
private void parseStatement() throws Lab3.AgainLab3.ParseException {
        Lab3.AgainLab3.Token token = consume();
        System.out.println("Parsed statement: " + token.match); //
Print the parsed statement
    }
```

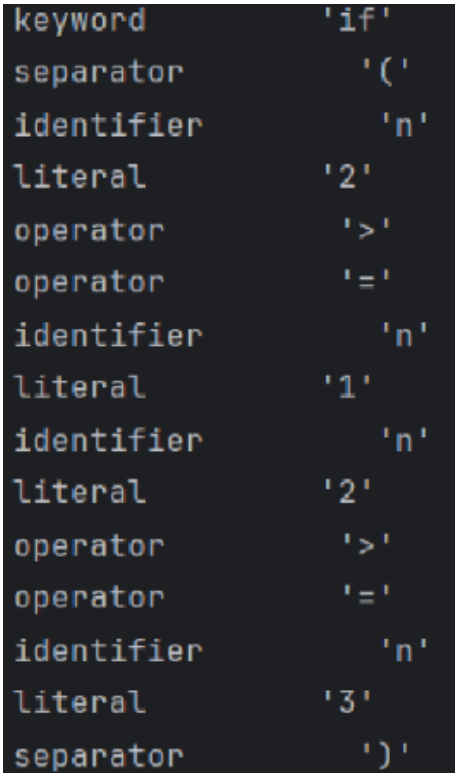
The consume() method fetches the current token from the list of tokens and moves to the next token by incrementing the currentTokenIndex. If the end of the token list is reached unexpectedly, a ParseException is thrown to indicate an error.

```
private Lab3.AgainLab3.Token consume() throws
Lab3.AgainLab3.ParseException {
        if (currentTokenIndex < tokens.size()) {
            return tokens.get(currentTokenIndex++);
        } else {
            throw new Lab3.AgainLab3.ParseException("Unexpected end of
input.");// Throw exception if end of input is reached
        }
    }
```

Finally, the main method calls the lexer function to generate tokens and then creates a Parser object with the generated tokens. The parse() method is invoked to initiate the parsing process. This sequence of actions ensures that the parser works on the output produced by the lexer, parsing the program's structure and handling individual statements.

```
public static void main(String[] args) {
        lexer();
        Lab3.AgainLab3.Parser parser = new
Lab3.AgainLab3.Parser(tokens);
        parser.parse();
    }
}
```

Output:



Conclusion:

This laboratory demonstrated fundamental concepts in computer science and programming language theory. This task provided understanding of formal grammars, including context-free grammars (CFGs) and regular expressions and learning of how these grammars define the syntax rules of programming languages and how parsers use them to recognize valid language constructs. During theoretical understanding, a great knowledge of various parsing techniques was taken into account. With this task, we've seen how that parsers play a crucial role in software development and language engineering and their real-world applications.