A Comprehensive Java Lexer and Parser with Visualizations

Group Details

Group ID - 36

Name	Enrollment
Abhishek Kumar Singh	21114003
Aryan Batra	21114019
Soham Singh	21114099

Problem Statement

PID - 4

Write a lexer and parser with LLVM compilation framework to compile basic java codes and
keep track of the basic constructs like loops, conditional branching, function, class, and object
etc. Also check for lexing and parsing errors. [Assigned TA: Tamal]

Deliverables:
A C++ code to design lexer and parser which works with LLVM compilation framework to
detect lexical and parsing errors if any otherwise show the tokens with type and also shows the
parse tree (use a dot file to represents parse tree) for a given Java code.

Introduction

This report provides a comprehensive overview of the parsing and lexical analysis utilizing the LLVM compilation framework , for java programming language, along with visualization capabilities. The project comprises four main components:

- Grammar specification - (`grammar.txt`)
- Token specification - (`valid_token.txt`)
- Lexer implementation - (`lex1.h`)
- Parser implementation - (`parse.cpp`)

Project Overview

The primary goal of this project is to develop a lexer and parser for java programming language. The lexer is responsible for converting the source code into a stream of tokens, while the parser analyzes the tokens to build a parse tree, representing the syntactic structure of the program. Additionally, the project includes functionalities for visualizing the parse tree using the Graphviz tool (copy paste `tree.dot` file here).

Components

- **Token Specification (`valid_token.txt`):**
  - The `valid_token.txt` file contains the formal specification of the programming language's tokens defines rules for lexical analysis.
- **Grammar Specification (`grammar.txt`):**
  - The `grammar.txt` file contains the formal specification of the programming language's grammar. It defines production rules for syntactic analysis.
- **Lexer Implementation (`lex1.h`):**
  - The `lex1.h` file contains the implementation of the lexer class, which processes the source code character by character and generates tokens based on the language's grammar rules.
  - The tokens recognized by lexer are written in `valid_token.txt`
  - It returns list(vector) of `tokens{lexeme, type, line_no}`.
- **Parser Implementation (`parse.cpp`):**
  - The `parse.cpp` file contains the implementation of the parser class, which utilizes the lexer to tokenize the source code and constructs a parse tree based on the grammar rules specified in `grammar.txt`. The

parser implements recursive descent parsing to analyze the syntactic structure of the program.

Lexer Module (`lex1.h`)

- **Functionality**: Converts input source code into a stream of tokens, identifying keywords, identifiers, literals, and other lexical elements.
- **Lexical Analysis Algorithm**:
  - Tokenizer scans the input character by character.
  - Rules define patterns for identifying tokens, such as keywords, identifiers, and literals.
  - Regular expressions or finite automata may underlie the tokenization process, efficiently recognizing token patterns.
- **Implementation**:
  - Implemented using DFA and longest prefix matching.
  - Utilizes a tokenizer to scan the input, categorizing characters into token types based on predefined rules.
- **Output:** It returns sequence of `tokens{lexeme, type, line_no}`.

Parser Module (parser.cpp)

- **Functionality**: Converts the sequence of tokens into parse tree.
- **Parsing Algorithm**:
  - Recursive Descent Parsing (RDP) is employed for top-down parsing.
  - The parser recursively explores the grammar rules, matching them against the token stream.
  - Each non-terminal in the grammar corresponds to a parsing function, aiding in the construction of the parse tree.
- **Implementation**:
  - Implemented using top-down recursive descent parser.
  - Enumerations: `NonTerminal`, `TokenType`.
  - Struct: `TreeNode`.

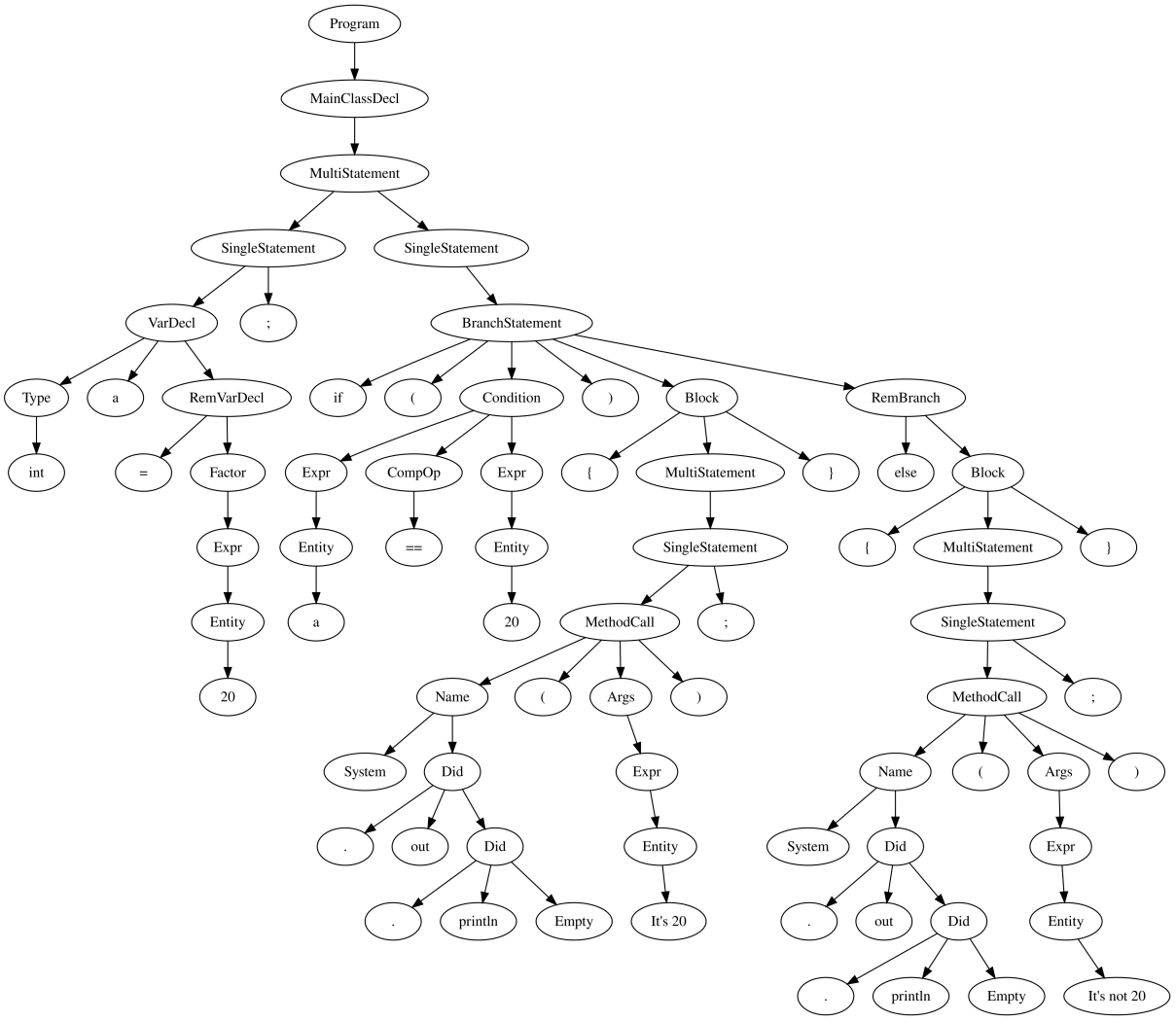
    - Classes: `Parser`.
    - Functions: `parse()`, `Program1()`, `MainClassDecl1()`, `MultiStatement1()`, `SingleStatement1()`, `VarDecl1()`, `AssignExpr1()`, `MethodCall1()`, `BranchStatement1()`, `RemBranch1()`, `WhileLoop1()`, `ForLoop1()`, `Block1()`, `Type1()`, `Factor1()`, `ObjConstruct1()`, `RemObj1()`, `Name1()`, `Did1()`, `BinOp1()`, `FrOp1()`, `BkOp1()`, `Expr1()`, `Condition1()`, `CompOp1()`, `ForInit1()`, `ForUpdate1()`, `Args1()`, `RemVarDecl1()`, `Entity1()`
- **DOT File Generation:**
    - Writes the parse tree to a DOT file for visualization using graph visualization tools like Graphviz.
    - Utilizes a depth-first traversal algorithm to traverse the parse tree and generate DOT file contents.
- **Output: Generate a `tree.dot` file containing parse tree**

Test 1 - Own Testcase

Input

```
public class input1 {
    public static void main(String[] args) {
        int a=20;
        if(a==20){
            System.out.println("It's 20");
        }
        else{
            System.out.println("It's not 20");
        }
    }
}
```

Output - parse tree



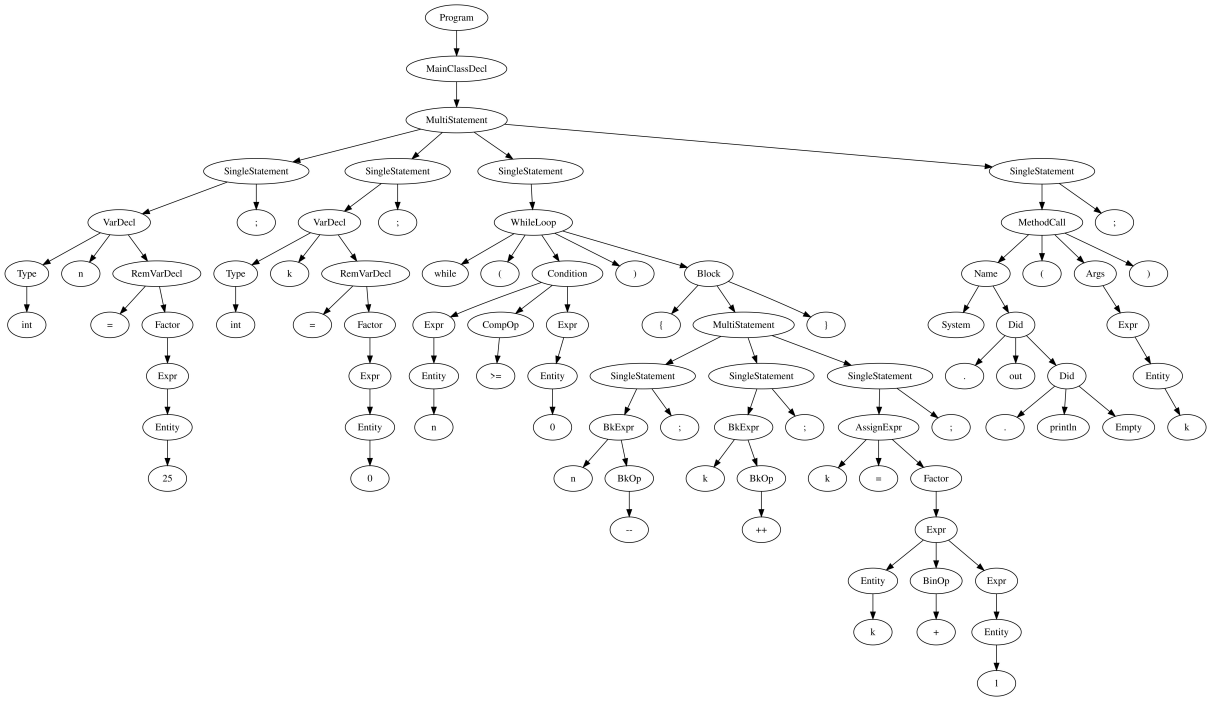
Test 2 - Own Testcase

Input

```
public class input2 {
    public static void main(String[] args) {
        int n=25;
        int k=0;
        while(n>=0){
            n--;
            k++;
            k=k+1;
        }
        System.out.println(k);
```

```
    }
}
```

Output - parse tree

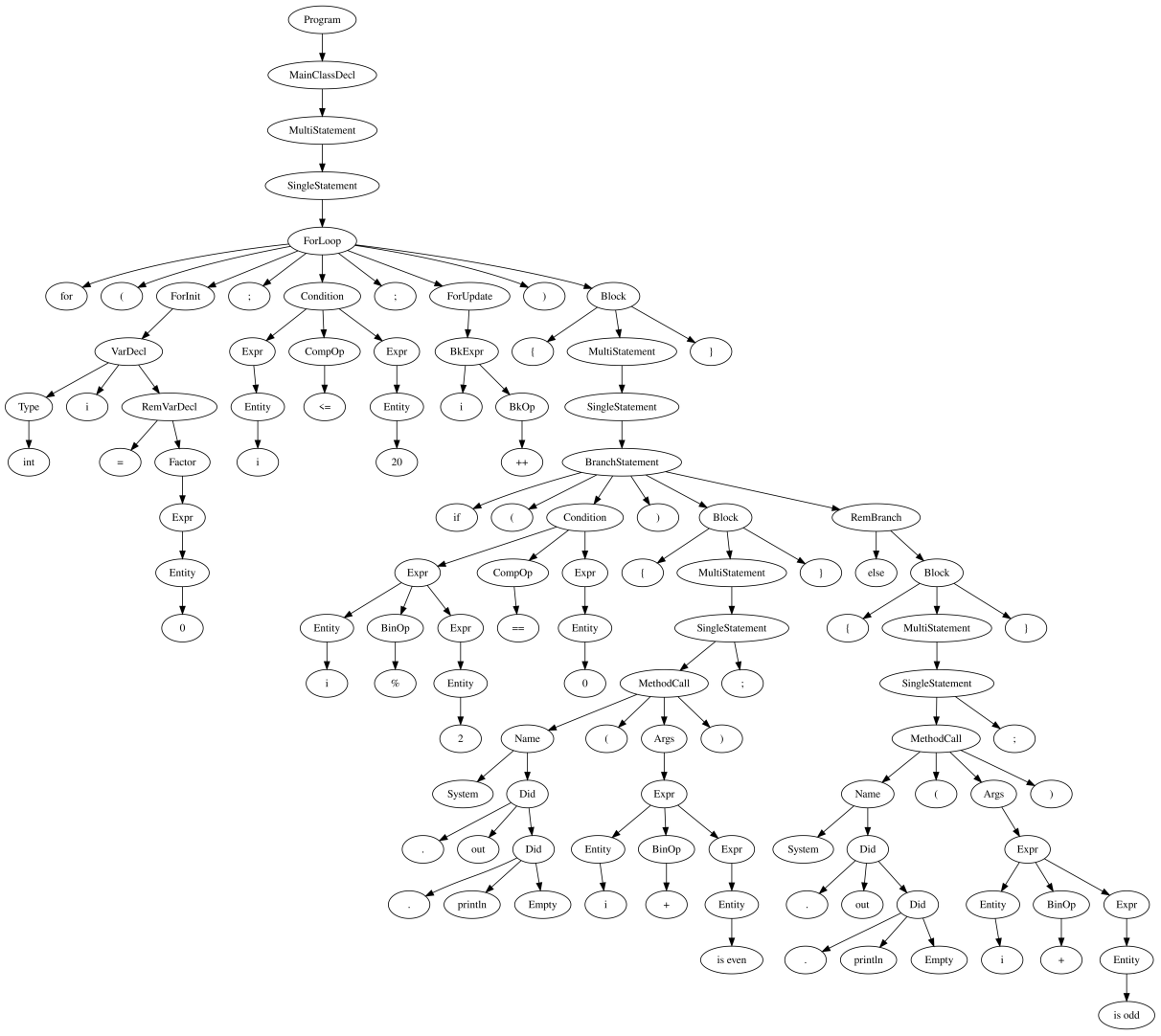


Test 3 - Given Testcase

Input

```
public class oddEvenDetector {
    public static void main(String[] args) {
        for (int i = 0; i <= 20; i++) {
            if (i % 2 == 0) {
                System.out.println(i + " is even");
            } else {
                System.out.println(i + " is odd");
            }
        }
    }
}
```

Output - parse tree



Weakness

- **Error Handling**: The current implementation lacks robust error handling mechanisms, leading to abrupt termination on encountering syntax errors. Future enhancements could incorporate better error recovery strategies.
- **Limited Grammar Support**: While the system supports a predefined grammar for Java-like languages, extending it to accommodate diverse grammars may require significant modifications.

Conclusion

In conclusion, our approach to parsing and lexical analysis, integrated with the LLVM compilation framework, offers a robust and efficient solution for language

processing tasks. While the system demonstrates strengths in terms of efficiency and flexibility, there exist areas for improvement, particularly in error handling and grammar support. Future work will focus on enhancing these aspects to further elevate the system's capabilities.