**Compiler Design Project**

**CSN - 352**

**Group – 19**

**Vishnu Sai Bikkumalla - 20114104**

**A Vashista Adithya - 20114001**

**Chakrala Sai Ram - 20114026**

**Problem Statement - 10**

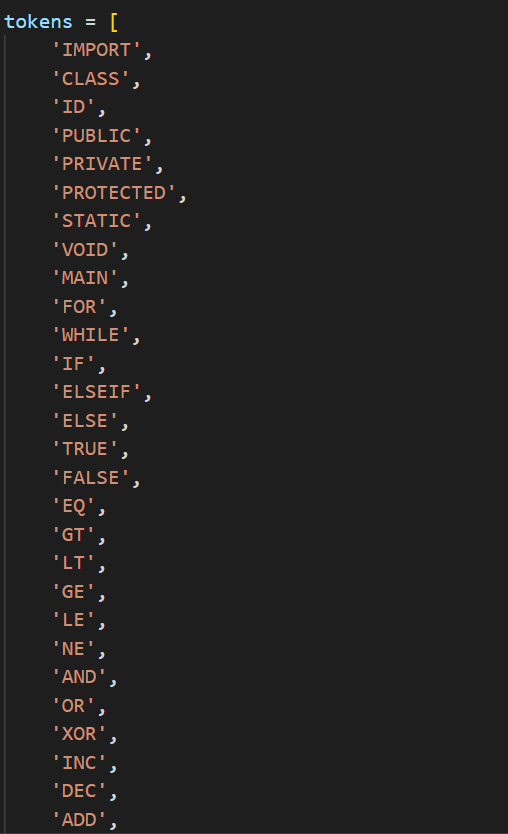
* Parser to accept the java code in python.

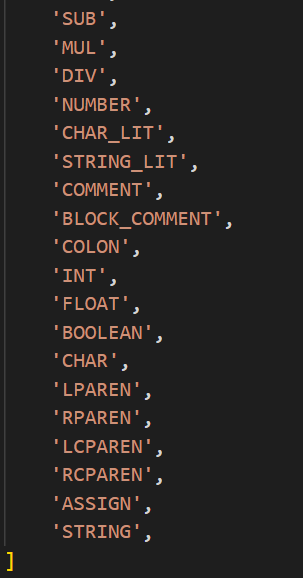
1. **Demonstration of code**

* **Define the lexer**:
* We will first design the lexer by first importing ply.lex
* It is a Python library that provides a lexer, also known as a scanner or tokenizer, for lexical analysis of input text.

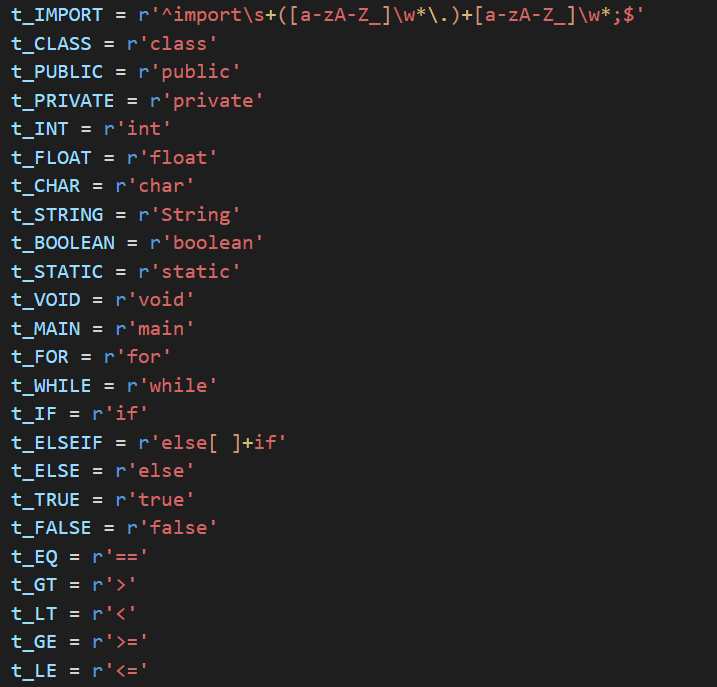
****

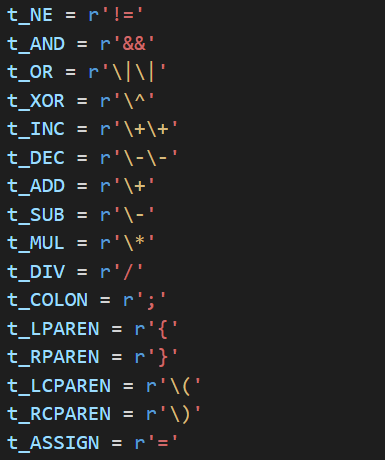
* Tokens:
* Below are the tokens.



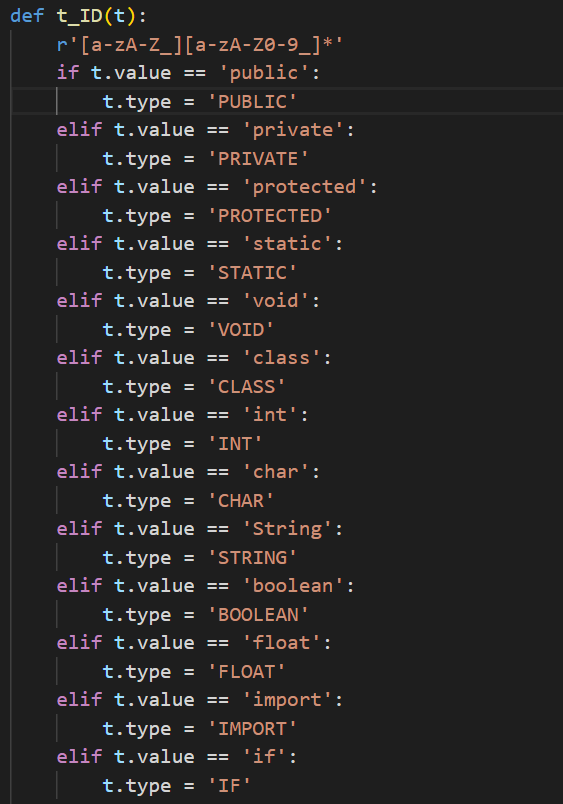
****

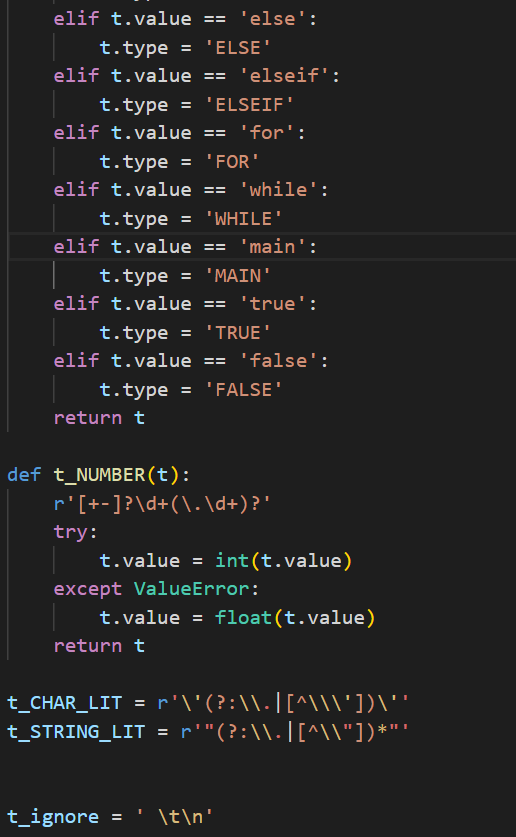
* Regular Expressions for the tokens:
* Below are the regular expressions for the tokens defined above.



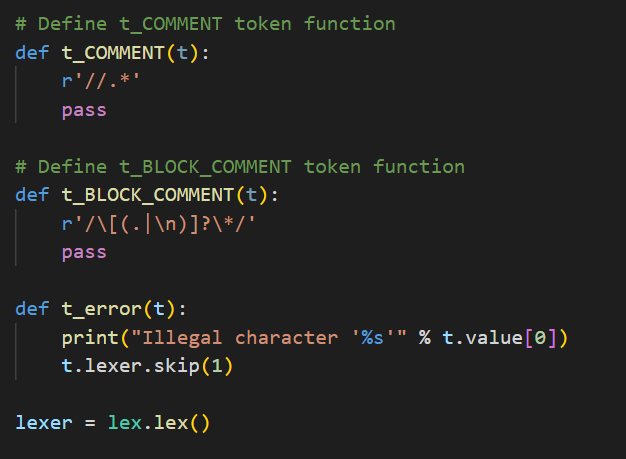


* Below are regular expressions for some specialized tokens.

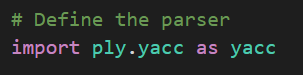




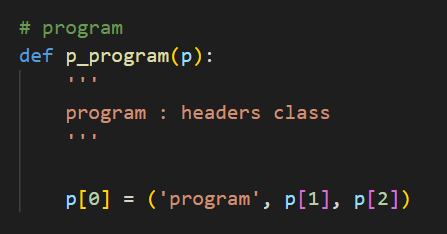
* Below are the regular expressions for comments, block of comments and errors respectively.



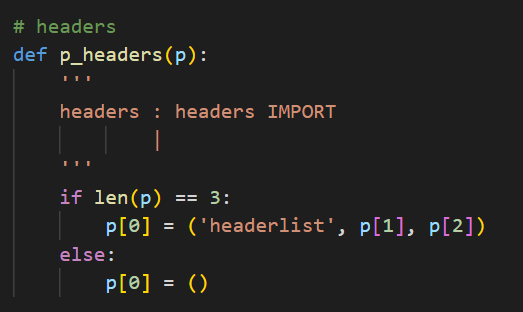
* And finally our lexer is prepared after feeding the above regular expressions to lex
* Define the parser
* We will start designing the parser by first importing ply.yacc.
* It is a Python library that provides a parser generator for building parsers that can interpret input text based on a formal grammar.



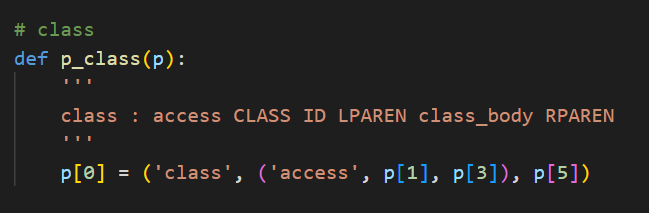
* Now we start building the parse tree doing syntax analysis starting from the below program.



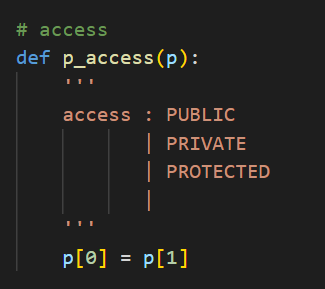
* So our program contains “headers” and “class”
* Below is the headers part of code.



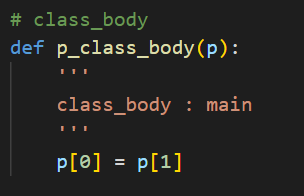
* And the class part is:



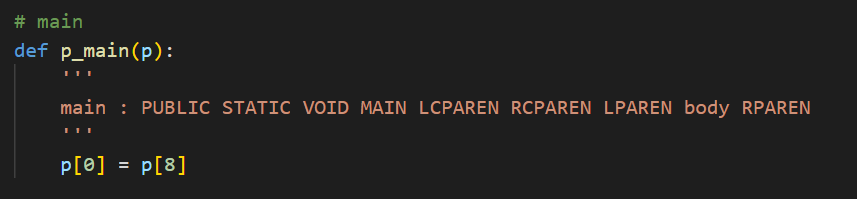
* Now in the class part, we have access and class\_body as non-terminals and others are tokens which are terminals.
* So access part of code is:



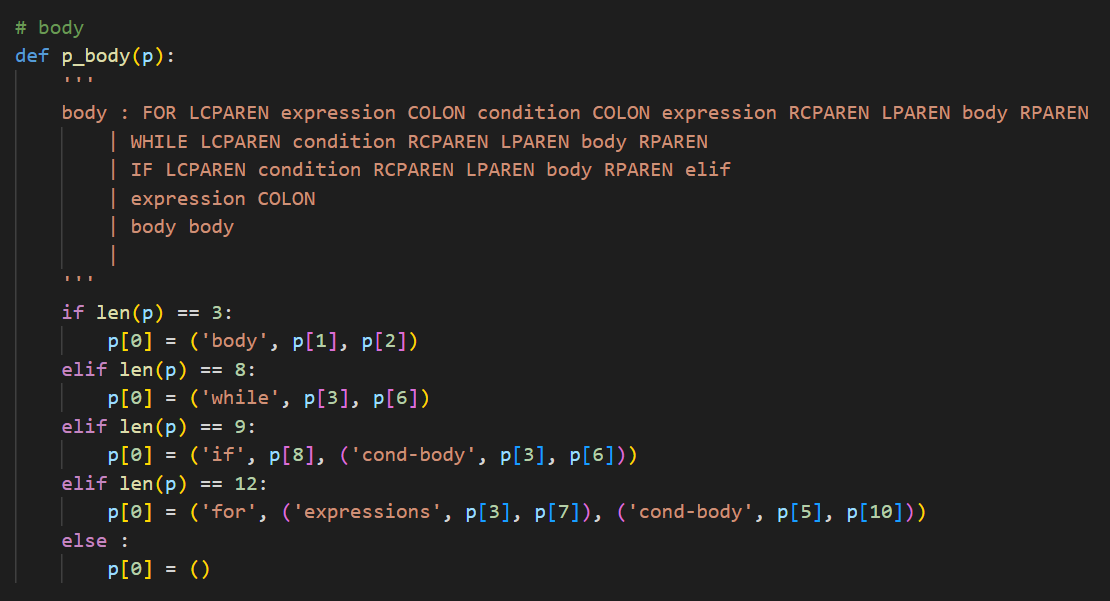
* And the class\_body part of code is:



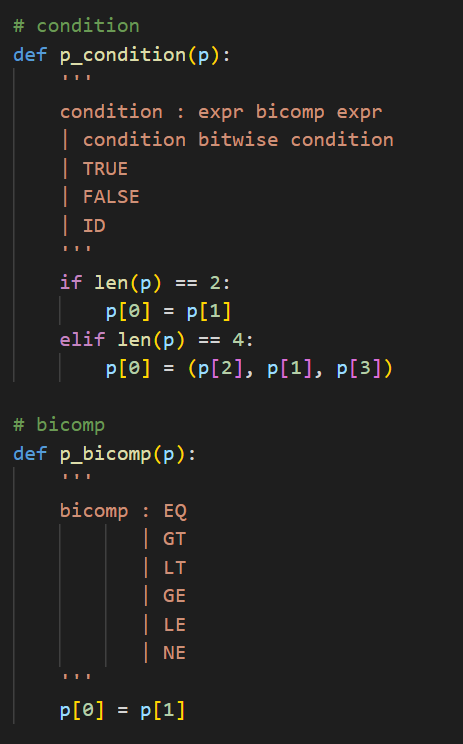
* In class\_body part, we have “main” as non-terminal. So let us expand that main part.
* Below is the code for main part:

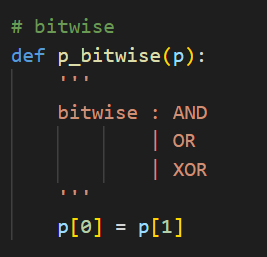


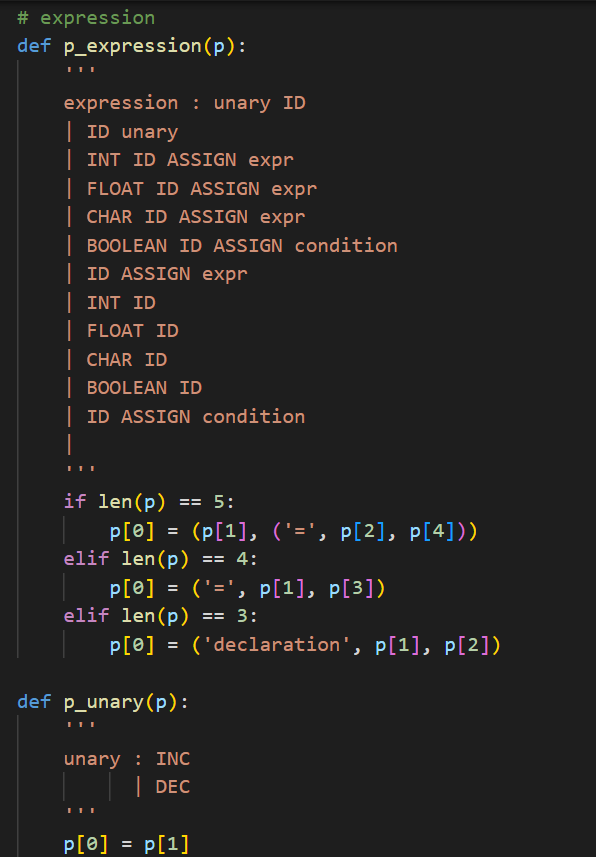
* Again in this main part, we have “body” as non-terminal. Below is the code for body part:

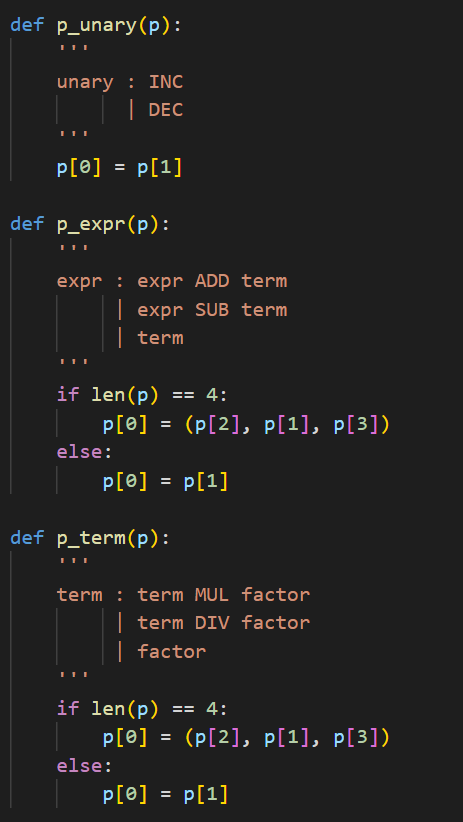


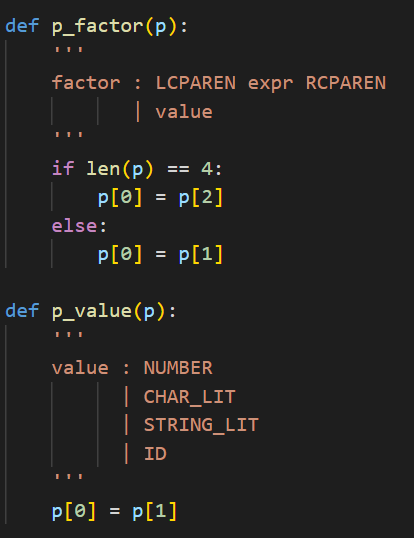
* Now expanding body gives non-terminals such as expression, condition, elif etc..
* So we get other code parts from them such as bicomp, bitwise (from condition expansion) and unary, expr, term, factor, value (from expression expansion). Below are the code parts:



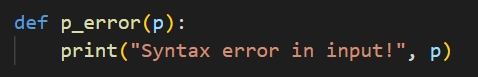








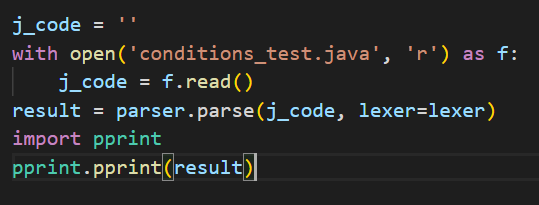
* Below is the code part for error



* And finally our parser will get ready after feeding the above CFG to the yacc.



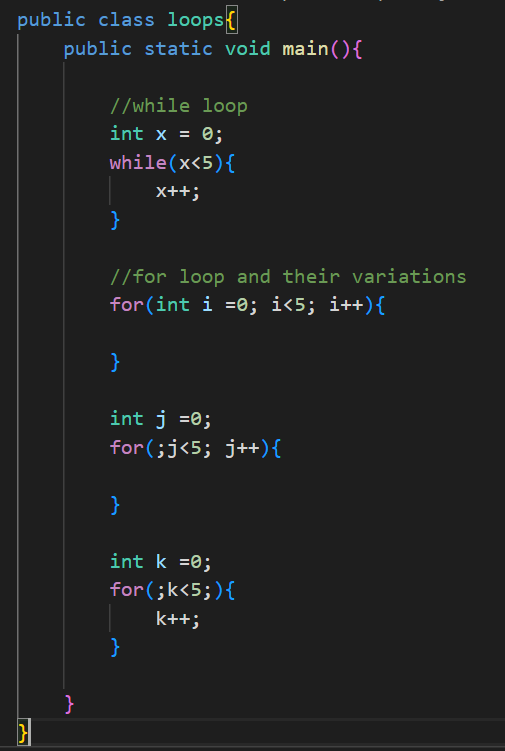
* **Input and Output**:
* Now the “j\_code” in the below code snippet is where we feed our input through f.



* So, now using this input from the j\_code and lexer(which we have prepared before) along with the above written parsing code gives us our final result which is stored in the “result”.
* That is, the syntax analysis is done, the parse tree is generated and the errors would be displayed if there are any.

1. **Test Cases**
2. Loop program :

Input : loop\_test.java

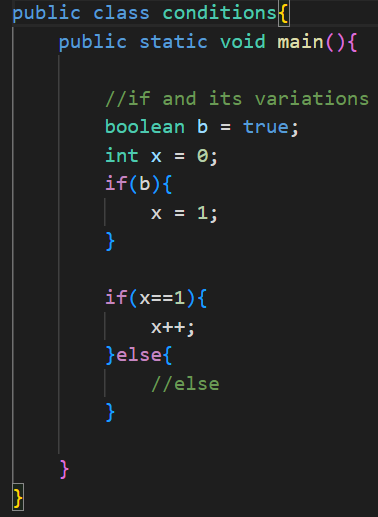


Output : Parse Tree-

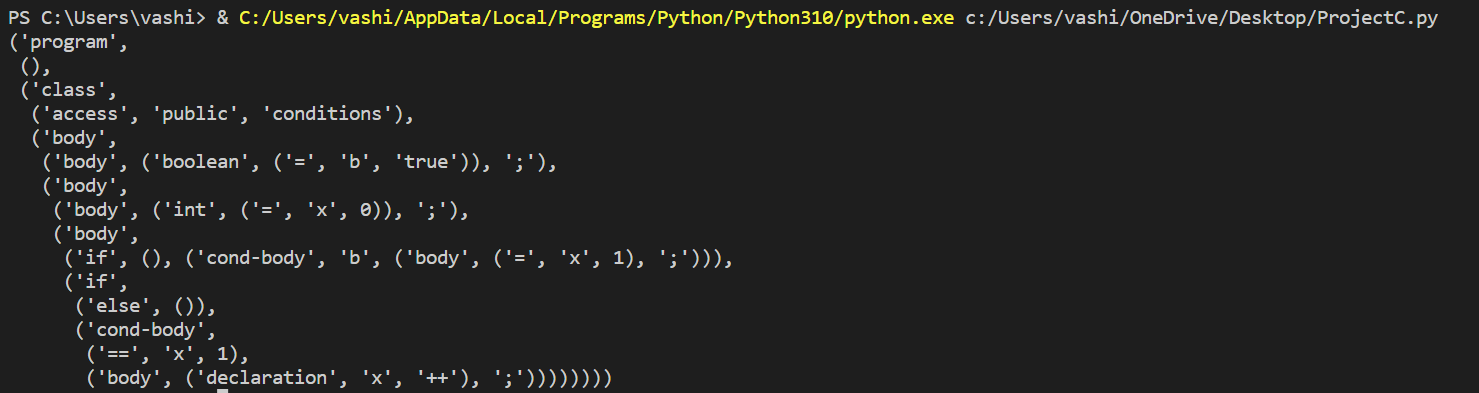


1. Conditions program :

Input : Conditions\_test.java

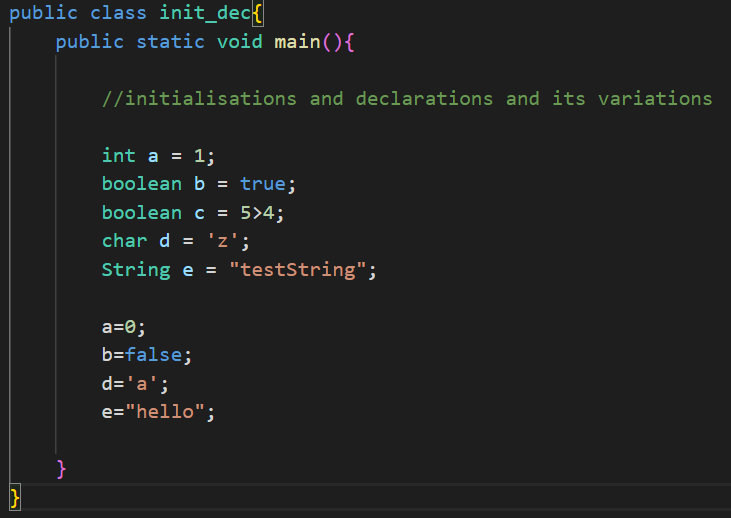


Output : Parse tree-



1. Declarations and Initializations program :

Input : Declarations.java

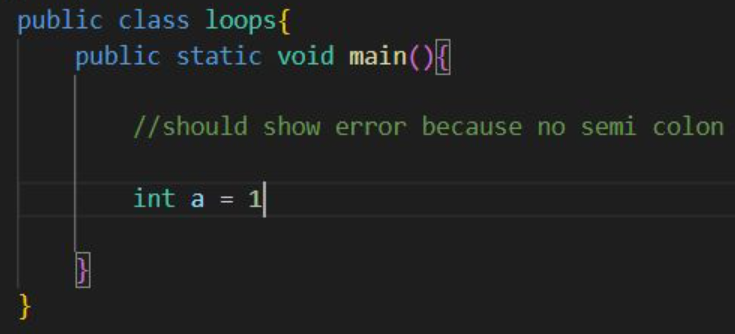
****

Output : Parse tree-

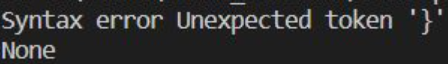
****

1. Error programs :

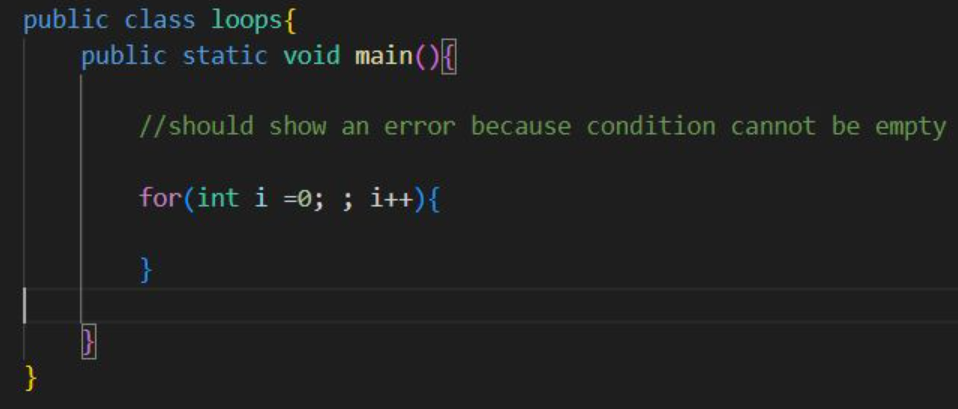
* Input : error\_test1.java



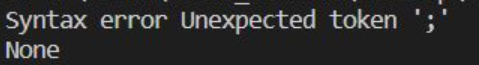
Output : Parse tree-



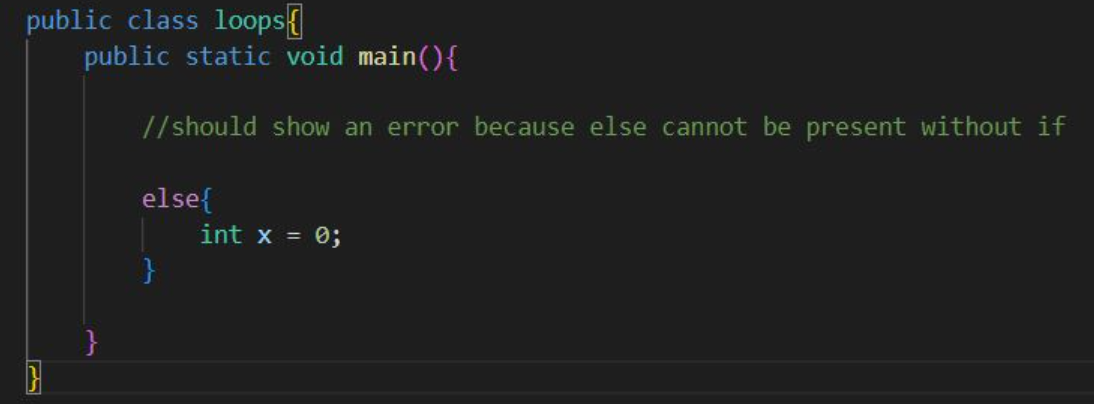
* Input : error\_test2.java



Output : Parse tree-



* Input : error\_test3.java



Output : Parse tree-

