



Department of Systemics
School Of Computer Science
UNIVERSITY OF PETROLEUM & ENERGY STUDIES,
DEHRADUN- 248007.
Uttarakhand

Compiler Design

Final Report On Compiler

Submitted By:

Divyansh Kumar

SAP- 500084995

Batch- 2

CSE CSF(Hons)

Submitted To:

Mr. Saurabh Shanu

COMPILER REPORT

OBJECTIVE: TO MAKE A COMPILER

THEORY:

COMPILER IS A PROGRAM THAT CONVERTS HIGH LEVEL LANGUAGE INTO LOW LEVEL LANGUAGE (COMPUTER UNDERSTANDABLE LANGUAGE.)

WORKING OF OUR COMPILER:

COMPILERS TRANSLATE THE CODE FROM A HIGHER-LEVEL LANGUAGE TO SOMETHING THAT THE COMPUTER CAN UNDERSTAND. IN ORDER TO ACCOMPLISH THIS, IT WILL READ THE INPUT (LEXICAL ANALYSIS, DONE BY THE LEXER) TO GENERATE A SERIES OF TOKENS. THESE TOKENS ARE THEN PASSED TO THE PARSER, WHICH GENERATES AN AST (ABSTRACT SYNTAX TREE) (SYNTACTIC ANALYSIS) WHICH REPRESENTS THE INPUT. IT THEN CONVERTS THAT AST INTO AN INTERMEDIATE REPRESENTATION OF CODE. THE COMPILER AT THIS POINT PERFORMS ONE OR MORE PASSES OVER THE INTERMEDIATE CODE PERFORMING OPTIMISATIONS. FINALLY, IT TAKES THAT OPTIMISED CODE AND EMITS MACHINE CODE FOR THE TARGET PLATFORM.

ALGORITHM:

STEP1: CREATE A NEW FILE OUTPUT.TXT WHERE OUR RESULT WILL BE SAVED.

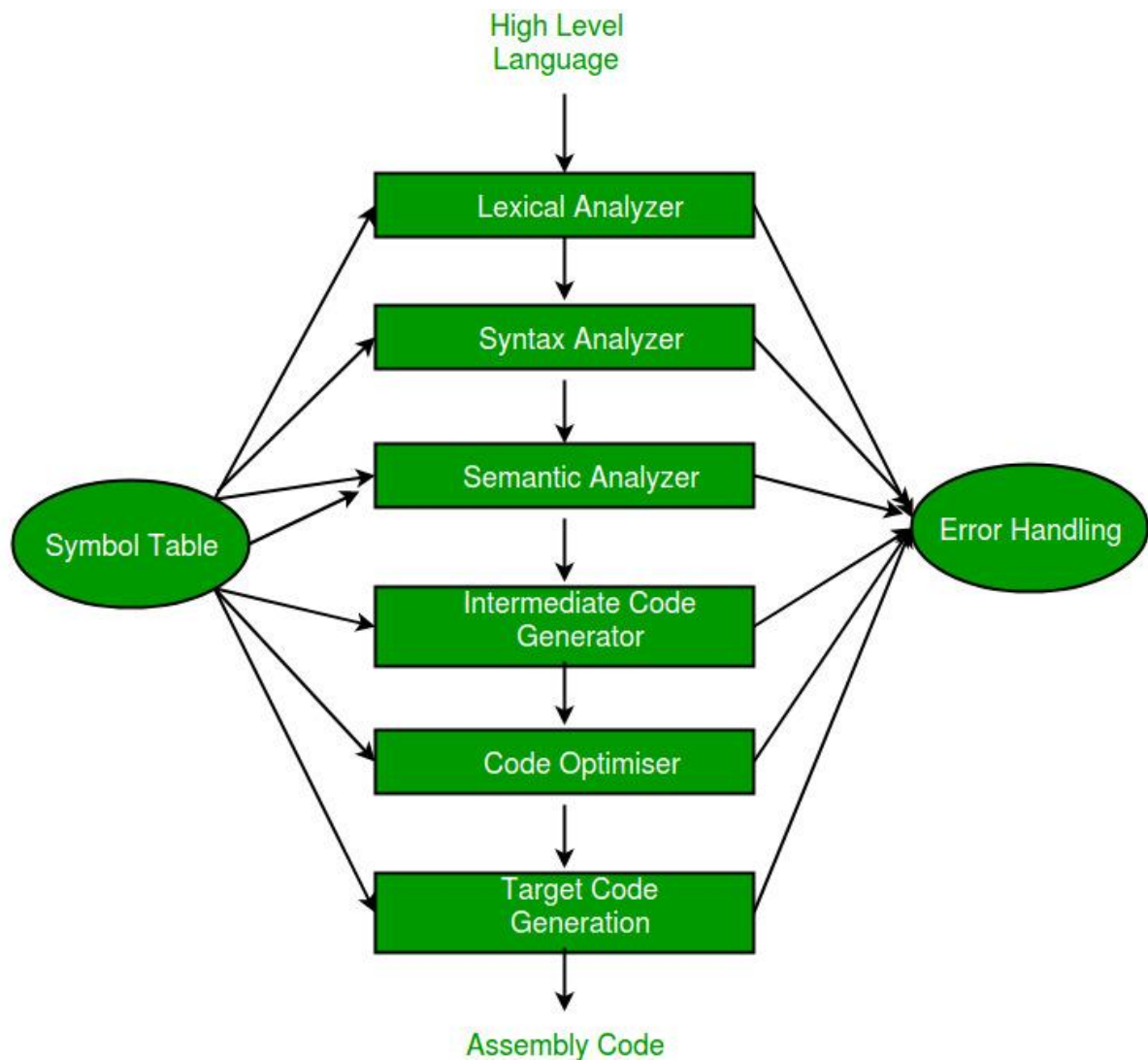
STEP2: GUI INTERFACE WHERE IN WE'LL GET THE USER'S INPUT OF STRING.

STEP3: SENDING THE ENTERED INPUT TO FUNCTION- TOKENIZER WHERE WE'LL PERFORM THE FIRST PHASE OF COMPILER I.E. LEXICAL ANALYSER AND SAVING THE RESULT IN OUTPUT.TXT FILE.

STEP4: CALLING FUNCTION PARSER TO GENERATE THE ABSTRACT SYNTAX TREE AND DISPLAYNG ERROR MESSAGE IF ANY.

STEP5: FURTHER GENERATING THE BINARY CODE FOR THE SAME INPUT OF STRING AND SAVING THAT IN OUPUT.TXT AND BINARY.BIN AS WELL.

PHASES OF COMPILER:



PHASES INCLUDED:

1. LEXICAL ANALYZER (TOKENIZATION)
2. SYNTAX ANALYZER (PARSER)
3. SEMANTIC ANALYZER (SEMANTICALLY VERIFIED PARSER)
4. BINARY CODE GENERATION

CODE:

```
import javax.swing.*;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;

public class Main {

    public static TreeMap<String, ArrayList<Character>> code = new
    TreeMap<>();

    public static void main(String[] args) throws FileNotFoundException {

        FileOutputStream f = new FileOutputStream("Output.txt");

        System.setOut(new PrintStream(f));
        JFrame frame = new JFrame("Divyansh's Compiler");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,300);
        JButton compileButton = new JButton("Compiler running");
        frame.getContentPane().add(compileButton);
        frame.setVisible(true);
        String input = JOptionPane.showInputDialog(null, "Enter your code:",
"(2+3)/4");

        System.out.println("Tokenization:\n=====\n");
        tokenizer(input);
        /* compiler tested for  tokenizer("(7+8)");
        tokenizer("(2+3)*4");
        tokenizer("10/(2+3)*4");
        parser("(2+4)");
        parser("+24"); // for this case shows error message
        parser("(2+3)*4");
        parser("10/(2+3)*4");*/
        System.out.println("-----
--\n\n\n");

        System.out.println("Parsing:\n-----\n");
        parser(input);

        System.out.println("\n\nBinary:");
        binaryGenerator(input);
```

```

        JOptionPane.showMessageDialog( null, "Output.txt and Binary.bin Files
Generated.",
        "Divyansh's Compiler", JOptionPane.PLAIN_MESSAGE );

    }
    //binary code generator
    public static void binaryGenerator(String input){

        String s = "";
        char[] ch = input.toCharArray();

        for(char c : ch){
            s = s + String.format("%8s", Integer.toBinaryString(c));
        }

        System.out.println(s);

        try {
            File myObj = new File("Binary.bin");
            if (myObj.createNewFile()) {
                System.out.println("File created: " + myObj.getName());
            } else {
                System.out.println("File already exists.");
            }

            Path path = Path.of("Binary.bin");

            Files.writeString(path, s);

            System.out.println("Binary File Generated.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

    }
    //syntax analyzer
    public static void parser(String input){
        try{

            char[] inputChar = input.toCharArray();

            Stack<String> stacks = new Stack<>();

            Character[] operators = new Character[]{'+', '-', '*', '/'};
            Character[] numbers = new Character[]{'0', '1', '2', '3', '4',
'5', '6', '7', '8', '9'};

```

```

ArrayList<Character> store = new ArrayList<>();
int storeCursor = 0;

int checker = 0;
for(int i = 0; i < input.length(); i++){

    if((Arrays.stream(operators).toList()).contains(inputChar[i]))
    {
        store.add(inputChar[i]);
        storeCursor++;
    }

    else if(inputChar[i] == '('){
        stacks.push(Character.toString(inputChar[i]));
        if (i == 0) {
        }
        else if
((Arrays.stream(operators).toList()).contains(inputChar[i-1])) {
            checker++;
        }

    }

    else if(inputChar[i] == ')'){
        stacks.push(Character.toString(inputChar[i]));
        if(i == inputChar.length - 1){
        }
        else if
((Arrays.stream(operators).toList()).contains(inputChar[i+1])) ){
            checker++;
        }

        if(checker != 0 && checker % 2 == 0 && !store.isEmpty() ){
            stacks.push(Character.toString(store.get(storeCursor-
1)));

            store.remove(storeCursor-1);
            storeCursor--;
            checker--;
        }
    }
    else{
        if(i == inputChar.length - 1){
            stacks.push(Character.toString(inputChar[i]));
        }
        else
if((Arrays.stream(operators).toList()).contains(inputChar[i+1])){
            while((Arrays.stream(operators).toList()).contains(input
Char[i+1])) {

```

```

        String insert = inputChar[i] +
Character.toString(inputChar[i+1]);
        stacks.push(insert);
        i++;
    }
    checker++;
}
else {
    stacks.push(Character.toString(inputChar[i]));
    checker++;
}

    if(checker != 0 && checker % 2 == 0){
        stacks.push(Character.toString(store.get(storeCursor-
1)));

        store.remove(storeCursor-1);
        storeCursor--;
        checker--;
    }

}
if(!store.isEmpty()){
    stacks.push(Character.toString(store.get(storeCursor-1)));
}

System.out.println("\nAbstract Syntax Tree for '" + input + "':
");

System.out.println("{\n" +
    "\ttype: 'Program',\n" +
    "\tbody: [{\n");

    Iterator<String> stackIterator = stacks.iterator();

    String tab = "\t\t";

    while(stackIterator.hasNext()){
        String s = stackIterator.next();

        if(checkParam(s.charAt(0)).equals("number")){
            System.out.println(tab + "type: '" +
checkParam(s.charAt(0)) + "'" + "\n" + tab + "Value: '" + s + "'");
        }
        else if(checkParam(s.charAt(0)).equals("operator")){
            System.out.print(tab + "type: '" + checkParam(s.charAt(0))
+ "'" + "\n" + tab);
            switch (s) {
                case "+" -> s = "add";

```

```

        case "-" -> s = "subtract";
        case "*" -> s = "multiply";
        case "/" -> s = "divide";
    }
    System.out.println("Operation: '" + s + "'");
}
else if(checkParam(s.charAt(0)).equals("paren") && s.charAt(0)
== '('){
    System.out.println(tab + "{\n");
    tab = tab + "\t";
}
else if(checkParam(s.charAt(0)).equals("paren") && s.charAt(0)
== ')'){
    tab = tab.substring(1);
    System.out.println(tab + "}\n");
}

}
tab = tab.substring(1);
System.out.println(tab + "]\n");

}
catch (Exception e){
    System.out.println("\n ----- ");
    System.out.println("| ERROR DETECTED!! |");
    System.out.println(" ----- \n");
}
}

public static String checkParam(Character value){
    Character[] numbers = new Character[]{'0', '1', '2', '3', '4', '5',
'6', '7', '8', '9'};

    Character[] alphabets = new Character[]{'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

    Character[] operators = new Character[]{'+', '-', '*', '/'};

    if(value.equals('(') || value.equals('')){
        return "parenthesis";
    }
    else if((Arrays.stream(numbers).toList()).contains(value)){
        return "number";
    }
}

```



```

        else if((Arrays.stream(alphabets).toList()).contains(value)){
            return "literal";
        }
        else if((Arrays.stream(operators).toList()).contains(value)){
            return "operator";
        }
        else{
            return "other";
        }
    }
}

//lexical analyzer
public static void tokenizer(String input){
    int cursor = 0;

    char[] inputChar = input.toCharArray();

    ArrayList<Character> ch = new ArrayList<>();
    ArrayList<Character> intChar = new ArrayList<>();
    ArrayList<Character> alphaChar = new ArrayList<>();
    ArrayList<Character> opChar = new ArrayList<>();

    while(cursor < input.length()){
        Character c = inputChar[cursor];

        if(checkParam(c).equals("parenthesis")){
            ch.add(c);
            code.put("paren", ch);

            cursor++;
        }

        if(checkParam(c).equals("number")){
            intChar.add(c);
            code.put("number", intChar);
            cursor++;
        }

        if(checkParam(c).equals("literal")){
            alphaChar.add(c);
            code.put("variable", alphaChar);
            cursor++;
        }

        if(checkParam(c).equals("operator")){
            opChar.add(c);
            code.put("operator", opChar);
            cursor++;
        }
    }
}

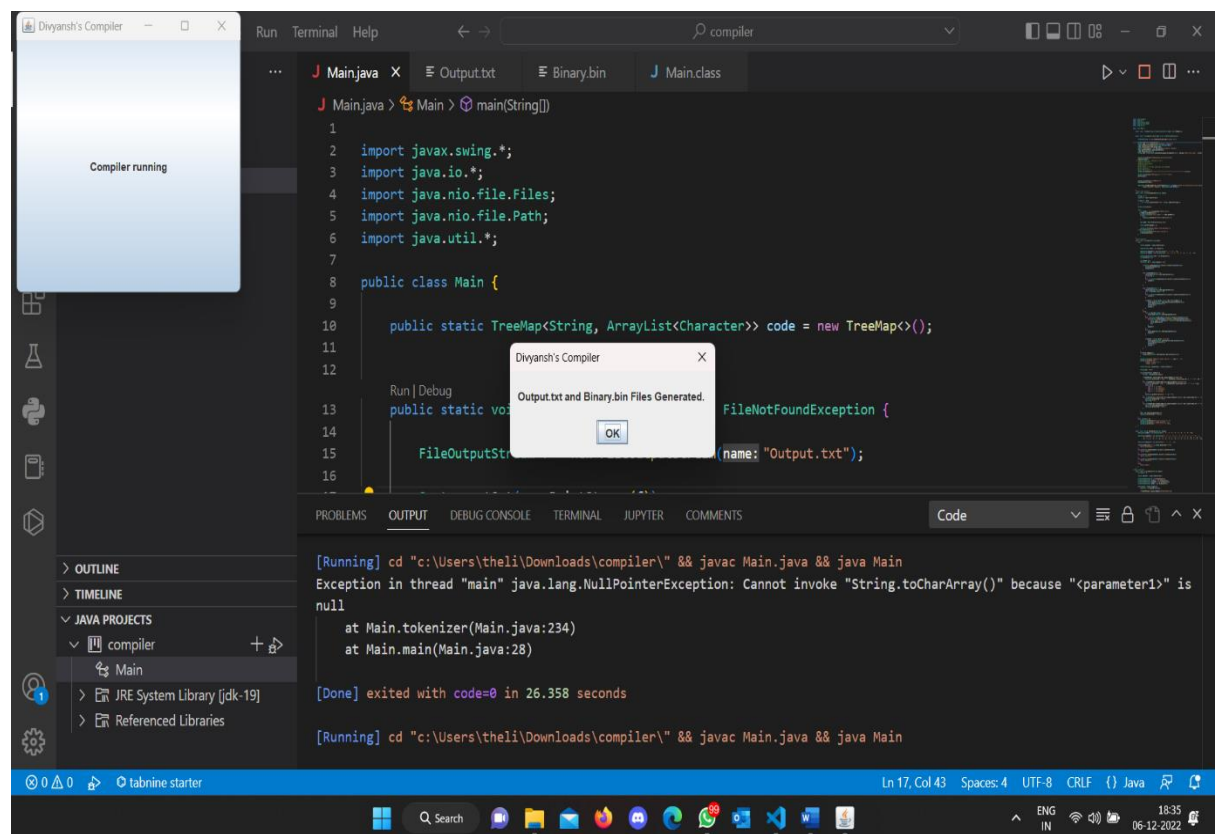
```

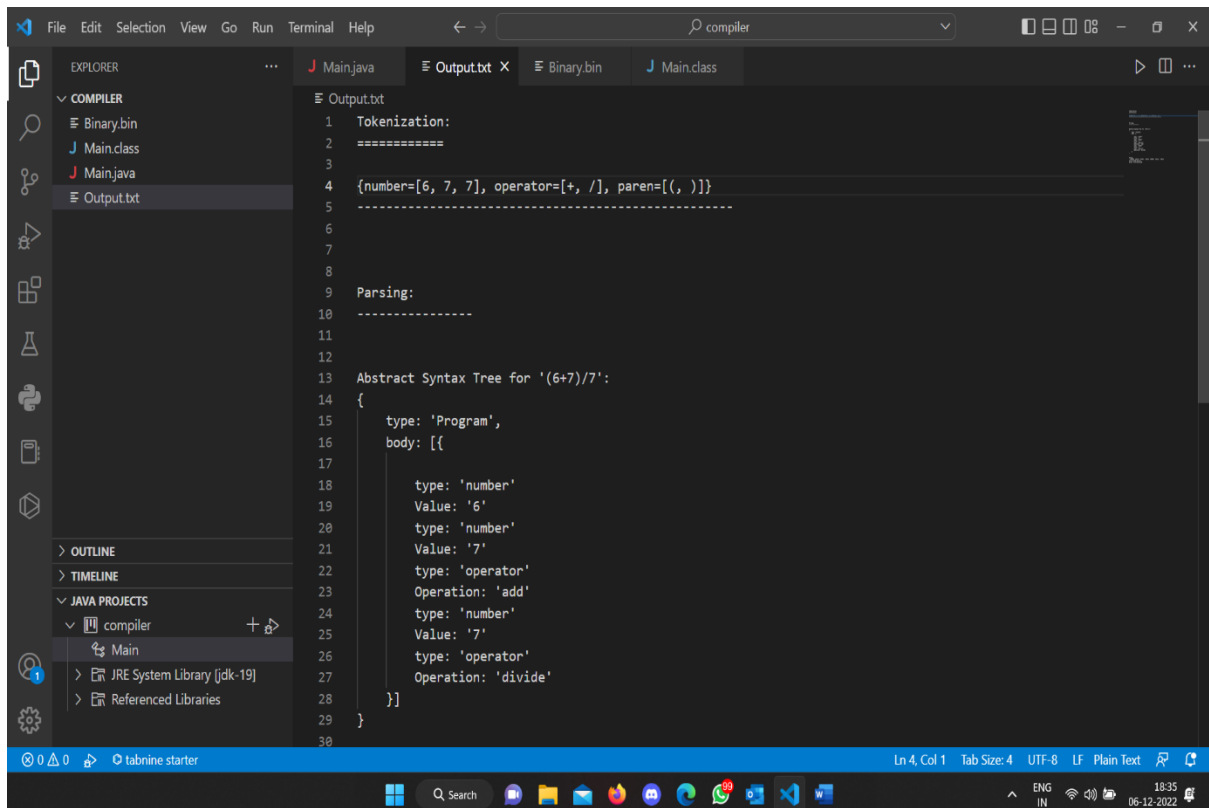
```

        if(checkParam(c).equals("other")){
            opChar.add(c);
            code.put("other", opChar);
            cursor++;
        }
    }
    System.out.println(code);
}
}

```

OUTPUT:





File Edit Selection View Go Run Terminal Help

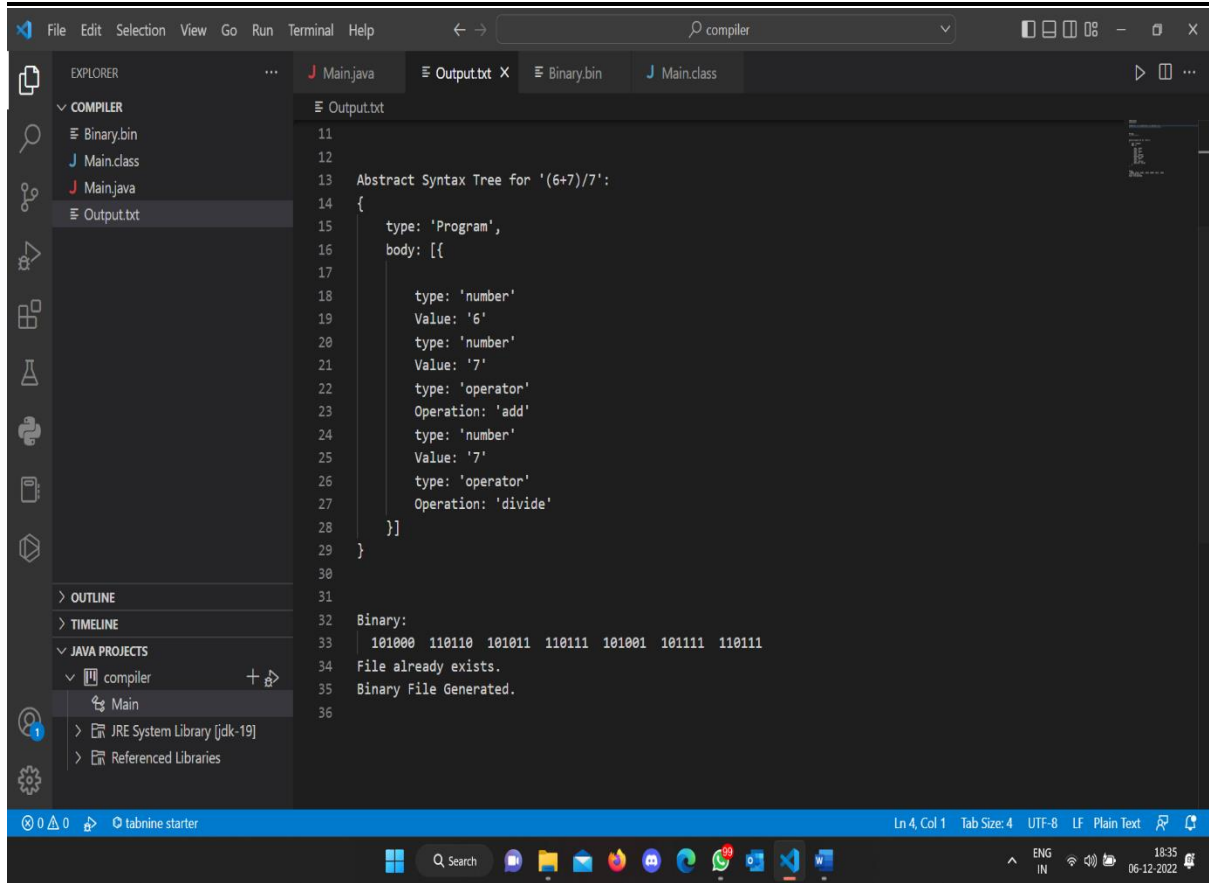
EXPLORER

- COMPILER
 - Binary.bin
 - Main.class
 - Main.java
 - Output.txt
- OUTLINE
- TIMELINE
- JAVA PROJECTS
 - compiler
 - Main
 - JRE System Library [jdk-19]
 - Referenced Libraries

Output.txt

```
1 Tokenization:
2 =====
3
4 {number=[6, 7, 7], operator=[+, /], paren=[(, )]}
5 -----
6
7
8
9 Parsing:
10 -----
11
12
13 Abstract Syntax Tree for '(6+7)/7':
14 {
15   type: 'Program',
16   body: [{
17     type: 'number'
18     Value: '6'
19     type: 'number'
20     Value: '7'
21     type: 'operator'
22     Operation: 'add'
23     type: 'number'
24     Value: '7'
25     type: 'operator'
26     Operation: 'divide'
27   }]
28 }
29
30
```

Ln 4, Col 1 Tab Size: 4 UTF-8 LF Plain Text



File Edit Selection View Go Run Terminal Help

EXPLORER

- COMPILER
 - Binary.bin
 - Main.class
 - Main.java
 - Output.txt
- OUTLINE
- TIMELINE
- JAVA PROJECTS
 - compiler
 - Main
 - JRE System Library [jdk-19]
 - Referenced Libraries

Output.txt

```
11
12
13 Abstract Syntax Tree for '(6+7)/7':
14 {
15   type: 'Program',
16   body: [{
17     type: 'number'
18     Value: '6'
19     type: 'number'
20     Value: '7'
21     type: 'operator'
22     Operation: 'add'
23     type: 'number'
24     Value: '7'
25     type: 'operator'
26     Operation: 'divide'
27   }]
28 }
29
30
31
32 Binary:
33 101000 110110 101011 110111 101001 101111 110111
34 File already exists.
35 Binary File Generated.
36
```

Ln 4, Col 1 Tab Size: 4 UTF-8 LF Plain Text

