الغلاف الخارجى للبحث

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| أولاً: البيانات الخاصة بالطالب | | | | | | | | | |
| **الفرقة الدراسية** | **الثالثة** | | | **التخصص** | | | **علوم حاسب** | | |
| **اسم القسم** | **CS** | | | | | | | | |
| **اسم المقرر** | **مترجمات** | | | | | | | | |
| **استاذ المقرر** | **د/ امال ابو طبل , د/ ميلاد** | | | | | | | | |
| ثانياً: البيانات الخاصة بالبحث | | | | | | | | | |
| **عنوان البحث** | **Scanner and Parser for Language 1** | | | | | | | | |
| **طبيعة المشاركة** | **بحث فردى** | | | | | **بحث جماعى**  \* | | | |
| **ارسال البحث** | **بواسطة البريد الالكتروني** | | | | | | | | |
| **اسماء الطلاب المشاركين فى البحث**  **(يكتب الاسم رباعيا)** | **م** | **الاسم رباعى** | | | | **رقم الجلوس** | | | **الرقم القومى** |
| **1** | **هشام أحمد حسن أحمد** | | | | **3239** | | | **29901242102912** |
| **2** | **محمود صلاح محمد جاد** | | | | **3216** | | | **29805250104631** |
| **3** | **محمود طارق محمد عبدالله** | | | | **3217** | | | **30001012105933** |
| **4** |  | | | |  | | |  |
| **5** |  | | | |  | | |  |
| **تاريخ الإرسال** | **9 / 6 / 2020** | | | | | | | | |
| ثالثاً: البيانات الخاصة بالكونترول | | | | | | | | | |
| **النتيجة** | | | **ناجح** | | **راسب** | | | | |
| **أعضاء لجنة تقييم البحث** | **الاسماء** | | | | | | | **التوقيع** | |
| **1** |  | | | | | |  | |
| **2** |  | | | | | |  | |
| **3** |  | | | | | |  | |

|  |  |
| --- | --- |
| **فى حالة عدم قبول البحث يرجى ذكر الأسباب** | * **..............................................................................................................................................** * **..............................................................................................................................................** * **..............................................................................................................................................** * **.............................................................................................................................................** |

**Contents**

|  |  |
| --- | --- |
| 1. **Introduction** 2. **Language description** 3. **Source code & Testing it** 4. **Project output results** 5. **Grammar** 6. **Parsing method used** 7. **First and Follow sets** 8. **LL(1) parsing table for the grammar** 9. **Parse Tree** 10. **Role of each group member** 11. **References** |  |

1. **Introduction**

We created Language 1, we used LL1 parser with table and stack and some other data structure and.

First, we have created the scanner and gave to every token some properties like token type, line number, and pattern,

Then, we created a simple BNF grammar and we handled the dangling-else problem with brackets, and also we used left-factoring and left-recursive grammar to convert BNF grammar to LL1 grammar to avoid ambiguous rules.

Then, we used the dictionary data structure to handle some errors like when a user uses a variable that does not exist or declares a variable more than ones, and also we handled syntax error with some description.

1. **Language description**

**Language 1,**

A program in this language consists of a main C function in which you can:  
• Declare integer, floating point and character variables.  
• Perform an assignment statement where the right-hand-side may be a  
constant, single variable or expression (including +,-,\*, / operators as well  
as () ).  
• Perform an if/if-else statement which may include any number of  
assignment statements and if/if-else statements in any order. The  
condition of the if/if-else statement may include >,<,>=,<=,==,!= and ().  
• In general, main function may include any number of assignment  
and if/if-else statements in any order and as described above.

This’s the language in a simple way without much details.

* if statement (else is not necessary for every if)

If (condition)

{statement ;}

Else

{statement ;}

* Assignment statement

Identifier = value or expression ;

* Declare statement

Datatype identifier = value or another identifier or expression;

* Datatype

Float, int, char

* Symbols = (, ), <, >, =, <=, >=, ==, !=, \*, /, +, -, }, {, ;, $

1. **Source code & Testing it**

Google drive link: https://drive.google.com/drive/folders/1-IPE2fzrnfda0jNWnFryonJlOjQwXpxh?usp=sharing

**Implemented source code**

1. Main Class
2. import Parser.Parser;  
   import Scanner.\*;  
     
   import java.io.File;  
   import java.io.FileNotFoundException;  
     
   public class Main {  
    public static void main(String[] args) throws Exception {  
    String path\_to\_file = "src/test case5.txt";  
    // Read code from the file  
    String code = *readFile*(path\_to\_file);  
     
    Parser parser = new Parser(new Scanner(code));  
    parser.start();  
     
    /\*  
    // testing the scanner  
    Scanner scanner = new Scanner(code);  
    while(scanner.hasNestToken()){  
    Token token = scanner.nextToken();  
    // these tokens and their types  
    System.out.print(token.getToken());  
    System.out.print(" "+token.getType());  
    System.out.println(" "+token.getLine());  
    }  
    \*/  
     
    }  
     
    private static String readFile(String path){  
    String code = "";  
    try {  
    File file = new File(path);  
    java.util.Scanner scanner = new java.util.Scanner(file);  
    int i = 1;  
    while (scanner.hasNextLine()){  
    String line = scanner.nextLine();  
    code += line + "\n" ;  
    ++i;  
    }  
    scanner.close();  
    }catch (FileNotFoundException e){  
    System.*out*.println(e);  
    }  
    return code;  
    }  
   }
3. Scanner Class
4. package Scanner;  
     
   import java.util.ArrayList;  
   import java.util.regex.Matcher;  
   import java.util.regex.Pattern;  
     
   */\*\*  
    \* what you will receive from this class? Are all tokens in the file you sent to the constructor as a string  
    \*/*public class Scanner {  
    // array list contains types of tokens and their patterns  
    private ArrayList<TokenInfo> tokensTypes;  
    //Code of strings to scan  
    private String file;  
    // counter to track line for each token  
    private int counter = 1;  
    public Scanner(String file){  
    this.tokensTypes = new ArrayList<TokenInfo>();  
    // remove space and new line from first and end of string  
    this.file = file + "$";  
    prepareTokensInfo();  
    }  
     
    private void prepareTokensInfo()  
    {  
    // For identifiers  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^(([a-zA-Z]([\_]|[a-zA-Z0-9])\*))"), TokenType.*IDENTIFIER*));  
    // For float value  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^((-)?[0-9]+(.)[0-9]+)"), TokenType.*FLOAT*));  
    // For integer value  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^((-)?[0-9]+)"), TokenType.*INTEGER*));  
    // For Char value  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^(\'(\\w|\\s)\')"), TokenType.*CHAR*));  
    // For Datatype  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^(int|float|char)"), TokenType.*DATATYPE*));  
    //others  
    for(String token : new String[] {">=", "<=", "==", "=", "!=","<", ">", ",","\\(", "\\)", "\\{", "\\}", "if", "else", "!", "\\+", "\\-", "\\/", "\\\*", "main", ";", "\\$"})  
    tokensTypes.add(new TokenInfo(Pattern.*compile*("^(" + token + ")"), TokenType.*TOKEN*));  
     
    }  
     
    public Token nextToken(){  
    // if it's new line so increase counter by 1  
    if(file.charAt(0) == '\n')  
    this.counter +=1;  
     
    // remove space and new line from first and end of string  
    file= file.trim();  
     
    //loop over RE to find or match current token  
    for(TokenInfo tokenInfo : tokensTypes)  
    {  
    Matcher matcher = tokenInfo.getPattern().matcher(file);  
     
    // if it's matches then remove this token from string file or code  
    if(matcher.find()){  
    String token = matcher.group();  
    file = matcher.replaceFirst("");  
     
    //Prepare tokens to parser or next phase  
    return PrepareToken(tokenInfo, token);  
    }  
    }  
     
    throw new IllegalStateException("Could not scan line " + counter);  
    }  
     
    */\*\*  
    \** ***@param*** *tokenInfo  
    \** ***@param*** *token  
    \** ***@return*** *Token  
    \*/* private Token PrepareToken(TokenInfo tokenInfo, String token){  
    //correct token type  
    if(token.equals("int") || token.equals("char") ||token.equals("float"))  
    return new Token(token, tokenInfo.getPattern(), TokenType.*DATATYPE*, counter);  
     
    if(token.equals("if") || token.equals("else") ||token.equals("main"))  
    return new Token(token, tokenInfo.getPattern(), TokenType.*TOKEN*, counter);  
     
    //check if token is char then skip first and last char ''  
    if(tokenInfo.getTokenType() == TokenType.*CHAR*)  
    return new Token(token.substring(1, token.length() - 1), tokenInfo.getPattern(), tokenInfo.getTokenType(), counter);  
     
    return new Token(token, tokenInfo.getPattern(), tokenInfo.getTokenType(), counter);  
    }  
     
    public boolean hasNestToken(){  
    return !this.file.isEmpty();  
    }  
   }

1. Token Class
2. package Scanner;  
   import java.util.regex.Pattern;  
   public class Token {  
    private String token;  
    private Pattern pattern;  
    private TokenType type;  
    private int line;  
    public Token(String token, Pattern pattern, TokenType type, int line)  
    {  
    this.pattern = pattern;  
    this.token = token;  
    this.type = type;  
    this.line = line;  
    }  
     
    public Pattern getPattern() {  
    return pattern;  
    }  
     
    public int getLine() {  
    return line;  
    }  
     
    public String getToken() {  
    return token;  
    }  
     
    public TokenType getType() {  
    return type;  
    }  
   }
3. TokenInfo Class
4. package Scanner;  
   import java.util.regex.Pattern;  
   public class TokenInfo {  
    private Pattern pattern;  
    private TokenType tokenType;  
    public TokenInfo(Pattern pattern, TokenType tokenType){  
    this.pattern = pattern;  
    this.tokenType = tokenType;  
    }  
     
    public Pattern getPattern() {  
    return pattern;  
    }  
     
    public TokenType getTokenType() {  
    return tokenType;  
    }  
     
    public void setTokenType(TokenType tokenType) {  
    this.tokenType = tokenType;  
    }  
   }

1. TokenType Enum
2. package Scanner;  
     
   public enum TokenType {  
    *TOKEN*,  
    *IDENTIFIER*,  
    *DATATYPE*,  
    *INTEGER*,  
    *CHAR*,  
    *FLOAT*}
3. Parser Class
4. package Parser;  
     
   import Scanner.\*;  
   import com.sun.org.apache.xpath.internal.objects.XString;  
   import javafx.util.Pair;  
     
   import java.util.Enumeration;  
   import java.util.Hashtable;  
   import java.util.Stack;  
   import java.util.Dictionary;  
     
   public class Parser {  
    private final Scanner scanner;  
    private Stack<String> stack;  
    private Token cur\_token;  
    private Dictionary<String, String> dictionary;  
    public static String *expecting*;  
    private boolean flag = true;  
     
    // constructor  
    public Parser(Scanner scanner){  
    this.scanner = scanner;  
    this.stack = new Stack<String>();  
    this.dictionary = new Hashtable<>();  
    }  
     
    */\*\*  
    \* function to start parsing the code  
    \*/* public void start() {  
    // check if the code is empty or not  
    if(!scanner.hasNestToken())  
    return;  
    // get the first token  
    cur\_token = scanner.nextToken();  
    stack.push("$");  
    stack.push("}");  
    // program is the start point  
    stack.push("program");  
    stack.push("{");  
    stack.push(")");  
    stack.push("(");  
    stack.push("main");  
    stack.push("int");  
    while(!stack.empty()){  
    match();  
    }  
    }  
     
    */\*\*  
    \* function to decide what next or what non terminal is gonna to release  
    \*/* private void match(){  
    // this is correct case where stack has only $ and last token is $  
    if(stack.peek().equals("$") && cur\_token.getToken().equals("$")){  
    System.*out*.println("SUCCESS");  
    stack.pop();  
    return;  
    }  
    else if (cur\_token.getType() == TokenType.*DATATYPE* && (stack.peek().equals("float") || stack.peek().equals("char") || stack.peek().equals("int")))  
    {  
    cur\_token = scanner.nextToken();  
    // add a new identifier to the dictionary  
    putNewIdentifier(cur\_token.getToken(), stack.peek(), cur\_token);  
    stack.pop();  
    }  
    // cases  
    else if (cur\_token.getToken().equals(stack.peek()))  
    {  
    cur\_token = scanner.nextToken();  
    stack.pop();  
    } else if (cur\_token.getType() == TokenType.*IDENTIFIER* && stack.peek().equals("id"))  
    {  
    // check if the identifier is declared or not before if not then that's an error  
    if(!checkIdentifier(cur\_token))  
    ERROR.*notDefinedError*(cur\_token);  
    cur\_token = scanner.nextToken();  
    stack.pop();  
    } else if ((cur\_token.getType() == TokenType.*INTEGER* || cur\_token.getType() == TokenType.*CHAR* || cur\_token.getType() == TokenType.*FLOAT*)  
    && stack.peek().equals("value"))  
    {  
    cur\_token = scanner.nextToken();  
    stack.pop();  
    }  
    else {  
    //System.out.println(stack.peek() +" " +cur\_token.getToken());  
    try {  
    // getting the top of the stack to expect the error  
    *expecting* = stack.peek();  
    // use the name of expression to call the appropriate function  
    Parser.class.getMethod(stack.pop()).invoke(this);  
    } catch (Exception e) {  
    // catch error in the code  
    ERROR.*syntaxError*(cur\_token);  
    }  
    }  
    }  
     
    */\*\*  
    \* these all the non terminals in the grammar and each non terminal has release of terminal or not  
    \*/* //1-  
    public void program(){  
    if(cur\_token.getToken().equals("if") || cur\_token.getType() == TokenType.*IDENTIFIER* ||  
    cur\_token.getType() == TokenType.*DATATYPE*)  
    stack.push("stmt\_seq");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //2-  
    public void stmt\_seq(){  
    if(cur\_token.getToken().equals("if") || cur\_token.getType() == TokenType.*IDENTIFIER* ||  
    cur\_token.getType() == TokenType.*DATATYPE*) {  
    stack.push("stmt\_seq2");  
    stack.push("stmt");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //3-  
    public void stmt\_seq2(){  
    if(!cur\_token.getToken().equals("if") && cur\_token.getType() != TokenType.*IDENTIFIER* &&  
    cur\_token.getType() != TokenType.*DATATYPE* && !cur\_token.getToken().equals("}") && !cur\_token.getToken().equals("$"))  
    ERROR.*syntaxError*(cur\_token);  
    else if(cur\_token.getToken().equals("}") || cur\_token.getToken().equals("$"))  
    Epsilon();  
     
    else  
    stack.push("stmt\_seq");  
    }  
    //4-  
    public void stmt(){  
    if(cur\_token.getToken().equals("if"))  
    stack.push("if\_stmt");  
    else if(cur\_token.getType() == TokenType.*IDENTIFIER*)  
    stack.push("assign\_stmt");  
    else if (cur\_token.getType() == TokenType.*DATATYPE*)  
    stack.push("declare\_stmt");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //5-  
    public void if\_stmt(){  
    if(cur\_token.getToken().equals("if")) {  
    stack.push("else\_part");  
    stack.push("}");  
    stack.push("stmt\_seq2");  
    stack.push("{");  
    stack.push(")");  
    stack.push("condition");  
    stack.push("(");  
    stack.push("if");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //6-  
    public void else\_part(){  
    if(cur\_token.getToken().equals("else")){  
    stack.push("}");  
    stack.push("stmt\_seq2");  
    stack.push("{");  
    stack.push("else");  
    }  
    else if (cur\_token.getToken().equals("if") || cur\_token.getType() == TokenType.*IDENTIFIER* || cur\_token.getType() == TokenType.*DATATYPE* || cur\_token.getToken().equals("$") || cur\_token.getToken().equals("}"))  
    Epsilon();  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //7-  
    public void condition(){  
    if(cur\_token.getType() == TokenType.*IDENTIFIER* ||  
    cur\_token.getType() == TokenType.*INTEGER* || cur\_token.getType() == TokenType.*FLOAT* || cur\_token.getType() == TokenType.*CHAR* || cur\_token.getToken().equals("(") ) {  
    stack.push("condition2");  
    stack.push("exp");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //8-  
    public void condition2(){  
    if(cur\_token.getToken().equals(")"))  
    Epsilon();  
    else if(cur\_token .getToken().equals("<") ||  
    cur\_token .getToken().equals(">") ||  
    cur\_token .getToken().equals("!=") ||  
    cur\_token .getToken().equals("<=") ||  
    cur\_token .getToken().equals(">=") ||  
    cur\_token .getToken().equals("==") )  
    {  
    stack.push("exp");  
    stack.push("comp\_sign");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
     
    }  
    //9-  
    public void comp\_sign(){  
    if(cur\_token.getToken().equals("<"))  
    stack.push("<");  
    else if (cur\_token.getToken().equals(">"))  
    stack.push(">");  
    else if (cur\_token.getToken().equals("!="))  
    stack.push("!=");  
    else if (cur\_token.getToken().equals("<="))  
    stack.push("<=");  
    else if (cur\_token.getToken().equals(">="))  
    stack.push(">=");  
    else if (cur\_token.getToken().equals("=="))  
    stack.push("==");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //10-  
    public void exp() {  
    if (cur\_token.getType() == TokenType.*IDENTIFIER* || cur\_token.getToken().equals("(")  
    || cur\_token.getType() == TokenType.*INTEGER* || cur\_token.getType() == TokenType.*FLOAT* || cur\_token.getType() == TokenType.*CHAR*) {  
    stack.push("exp2");  
    stack.push("term");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //11-  
    public void exp2(){  
    if(cur\_token.getToken().equals("+") | cur\_token.getToken().equals("-")) {  
    stack.push("exp2");  
    stack.push("term");  
    stack.push("add\_op");  
    }  
    else if(cur\_token.getToken().equals(")") ||  
    cur\_token.getToken().equals("<") ||  
    cur\_token.getToken().equals(">") ||  
    cur\_token.getToken().equals("!=") ||  
    cur\_token.getToken().equals("<=")||  
    cur\_token.getToken().equals(">=")||  
    cur\_token.getToken().equals("==")||  
    cur\_token.getToken().equals(";")) {  
    Epsilon();  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //12-  
    public void add\_op(){  
    if(cur\_token.getToken().equals("+"))  
    stack.push("+");  
    else if(cur\_token.getToken().equals("-"))  
    stack.push("-");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //13-  
    public void term(){  
    if(cur\_token.getToken().equals("(") ||  
    cur\_token.getType() == TokenType.*INTEGER* ||  
    cur\_token.getType() == TokenType.*FLOAT* ||  
    cur\_token.getType() == TokenType.*CHAR* ||  
    cur\_token.getType() == TokenType.*IDENTIFIER*) {  
    stack.push("term2");  
    stack.push("factor");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //14-  
    public void term2(){  
    if(cur\_token.getToken().equals("\*") || cur\_token.getToken().equals("/")){  
    stack.push("term2");  
    stack.push("factor");  
    stack.push("mul\_op");  
    } else if(cur\_token.getToken().equals("<")||  
    cur\_token.getToken().equals(">")||  
    cur\_token.getToken().equals("!=")||  
    cur\_token.getToken().equals("<=")||  
    cur\_token.getToken().equals(">=")||  
    cur\_token.getToken().equals("==")||  
    cur\_token.getToken().equals(";")||  
    cur\_token.getToken().equals("+")||  
    cur\_token.getToken().equals(")")||  
    cur\_token.getToken().equals("-"))  
    Epsilon();  
     
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //15-  
    public void mul\_op(){  
    if(cur\_token.getToken().equals("\*"))  
    stack.push("\*");  
    else if(cur\_token.getToken().equals("/"))  
    stack.push("/");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //16-  
    public void factor(){  
    if (cur\_token.getToken().equals("(")){  
    stack.push(")");  
    stack.push("exp");  
    stack.push("(");  
    }  
    else if(cur\_token.getType() == TokenType.*IDENTIFIER*)  
    stack.push("id");  
    else if(cur\_token.getType() == TokenType.*CHAR* ||  
    cur\_token.getType() == TokenType.*INTEGER* ||  
    cur\_token.getType() == TokenType.*FLOAT*) {  
    stack.push("value");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //17-  
    public void declare\_stmt(){  
    if(cur\_token.getType() == TokenType.*DATATYPE*) {  
    stack.push(";");  
    stack.push("x\_stmt");  
    stack.push("id");  
    stack.push("datatype");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //18-  
    public void x\_stmt(){  
    if(cur\_token.getToken().equals("=")){  
    stack.push("exp");  
    stack.push("=");  
    }  
    else if(cur\_token.getToken().equals(";"))  
    Epsilon();  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //19-  
    public void assign\_stmt(){  
    if(cur\_token.getType() == TokenType.*IDENTIFIER*) {  
    stack.push(";");  
    stack.push("exp");  
    stack.push("=");  
    stack.push("id");  
    }  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
    //20-  
    public void datatype(){  
    if(cur\_token.getToken().equals("int"))  
    stack.push("int");  
    else if(cur\_token.getToken().equals("float"))  
    stack.push("float");  
    else if(cur\_token.getToken().equals("char"))  
    stack.push("char");  
    else  
    ERROR.*syntaxError*(cur\_token);  
    }  
     
    */\*\*  
    \* check if the identifier is exist or not  
    \*/* private boolean checkIdentifier(Token token){  
    if(dictionary.get(token.getToken()) == null)  
    return false;  
    return true;  
    }  
     
    private void putNewIdentifier(String id, String datatype, Token token){  
    if(!checkIdentifier(token)){  
    dictionary.put(id, datatype);  
    }  
    else  
    ERROR.*definedError*(token);  
    }  
    */\*\*  
    \* just function to do nothing  
    \*/* public void Epsilon(){  
    }  
   }
5. Error Class
6. package Parser;  
   import Scanner.\*;  
   import sun.security.krb5.internal.PAData;  
     
   public abstract class ERROR {  
    // syntax error that's generated from the parser or bnf grammar  
    public static void syntaxError(Token token){  
    if(Parser.*expecting*.length() > 2 )  
    {  
    if(Parser.*expecting*.equals("stmt\_seq2"))  
    Parser.*expecting* = "}";  
    else if (Parser.*expecting*.equals("else\_part"))  
    Parser.*expecting* = "if or variable or datatype or }";  
    else if (Parser.*expecting*.equals("condition2"))  
    Parser.*expecting* = ")";  
    else if (Parser.*expecting*.equals("exp2"))  
    Parser.*expecting* = ") or comparision sign or ;";  
    else if (Parser.*expecting*.equals("term2"))  
    Parser.*expecting* = ") or comparision sign or ; or + or -";  
    else if (Parser.*expecting*.equals("x\_stmt"))  
    Parser.*expecting* = ";";  
    }  
    if(Parser.*expecting*.equals("id"))  
    Parser.*expecting* = "variable";  
    throw new IllegalStateException("Could not parse line " + token.getLine() + " at "+  
    "\"" + token.getToken() + "\"" + ", expected " + Parser.*expecting* + " (syntax error)");  
    }  
     
    // if the identifier is not defined before and user is using it.  
    public static void notDefinedError(Token token){  
    throw new IllegalStateException("Could not parse line " + token.getLine() +", \"" + token.getToken() + "\"" + " is not defined");  
    }  
     
    // if the identifier is already exist in the memory(dictionary) and user is declaring it again.  
    public static void definedError(Token token){  
    throw new IllegalStateException("Could not parse line " + token.getLine() +", \"" + token.getToken() + "\"" + " is already defined");  
    }  
     
   }

**Source code files used for testing**

1. Test case1

int main(){

int x = 10;

float y = 5.5;

x = y + ((5 \* 10)-(5-6 \* 8) / 2) - 3;

if(x > y){

if((x + y) == 0){

x = 1;

}else {

x = 0;

}

}else{

y = x;

}

}

1. Test case2

int main(){

int x = 5;

int y = 6;

if(5 + 6{

x = 5 + 7;

}

}

1. Test case3

int main(){

int x = 5;

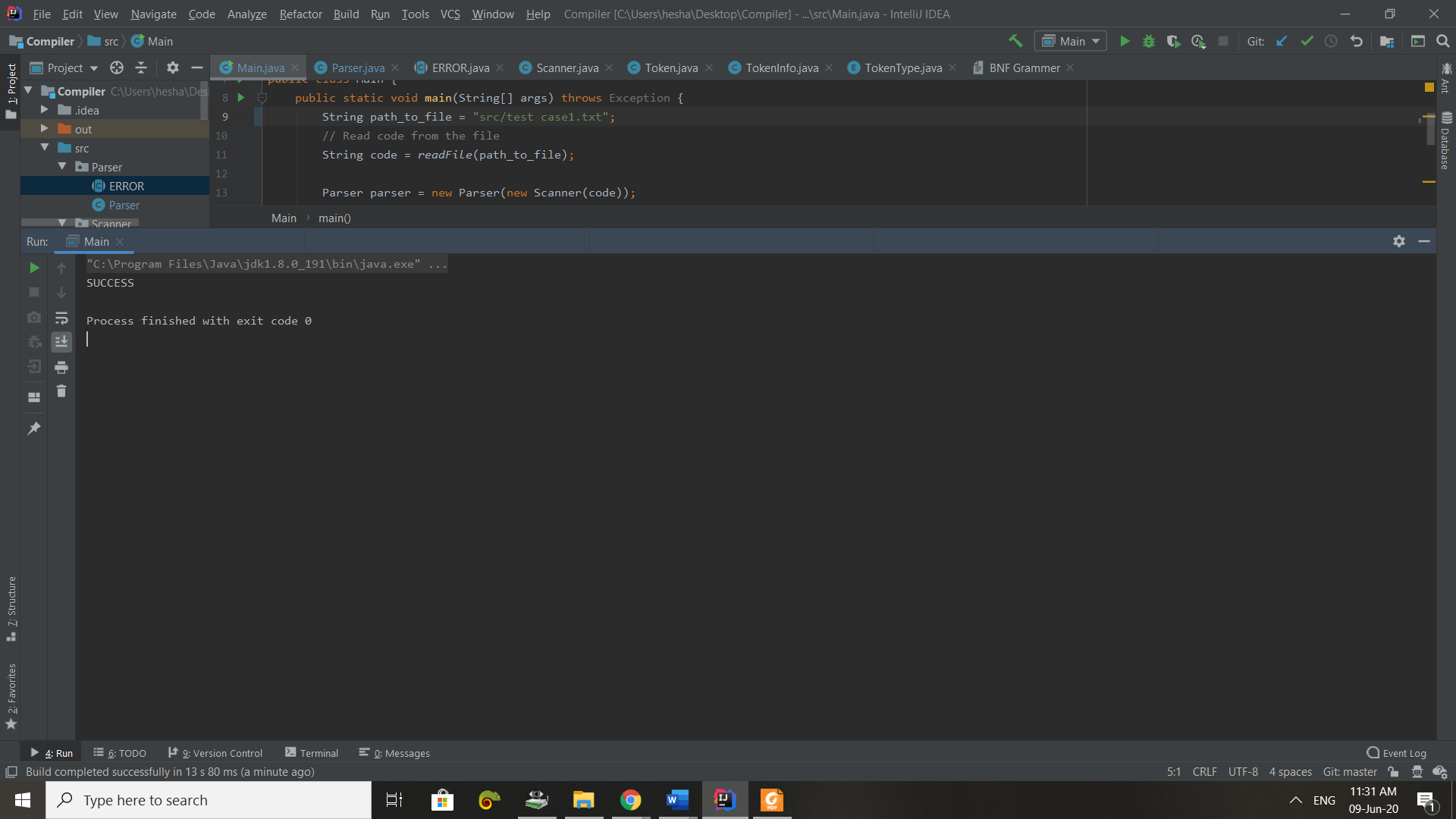
int x = 20;

}

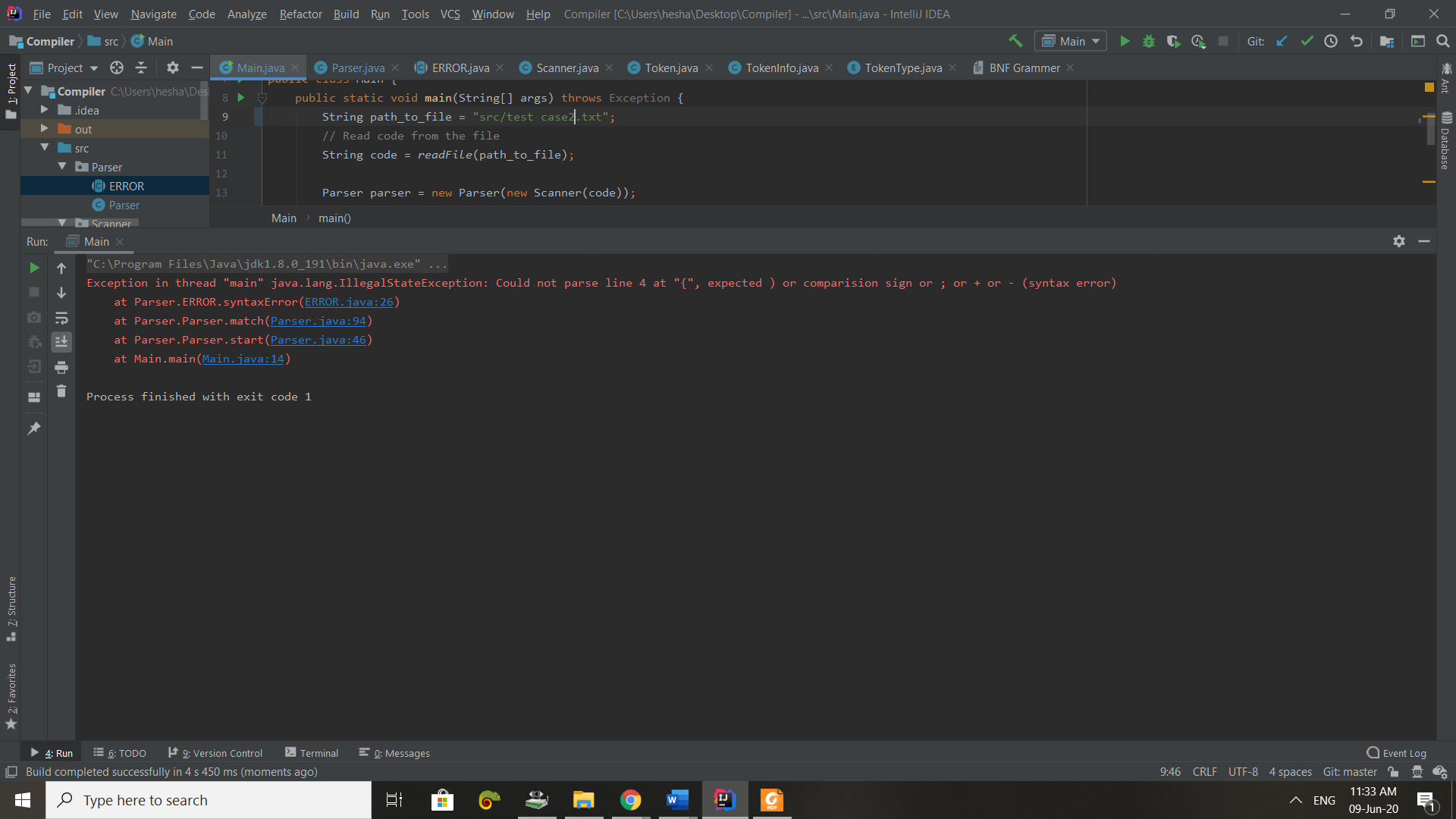
1. **Project output results**

* **Screenshots of Scanner and Parser output(success/error)**

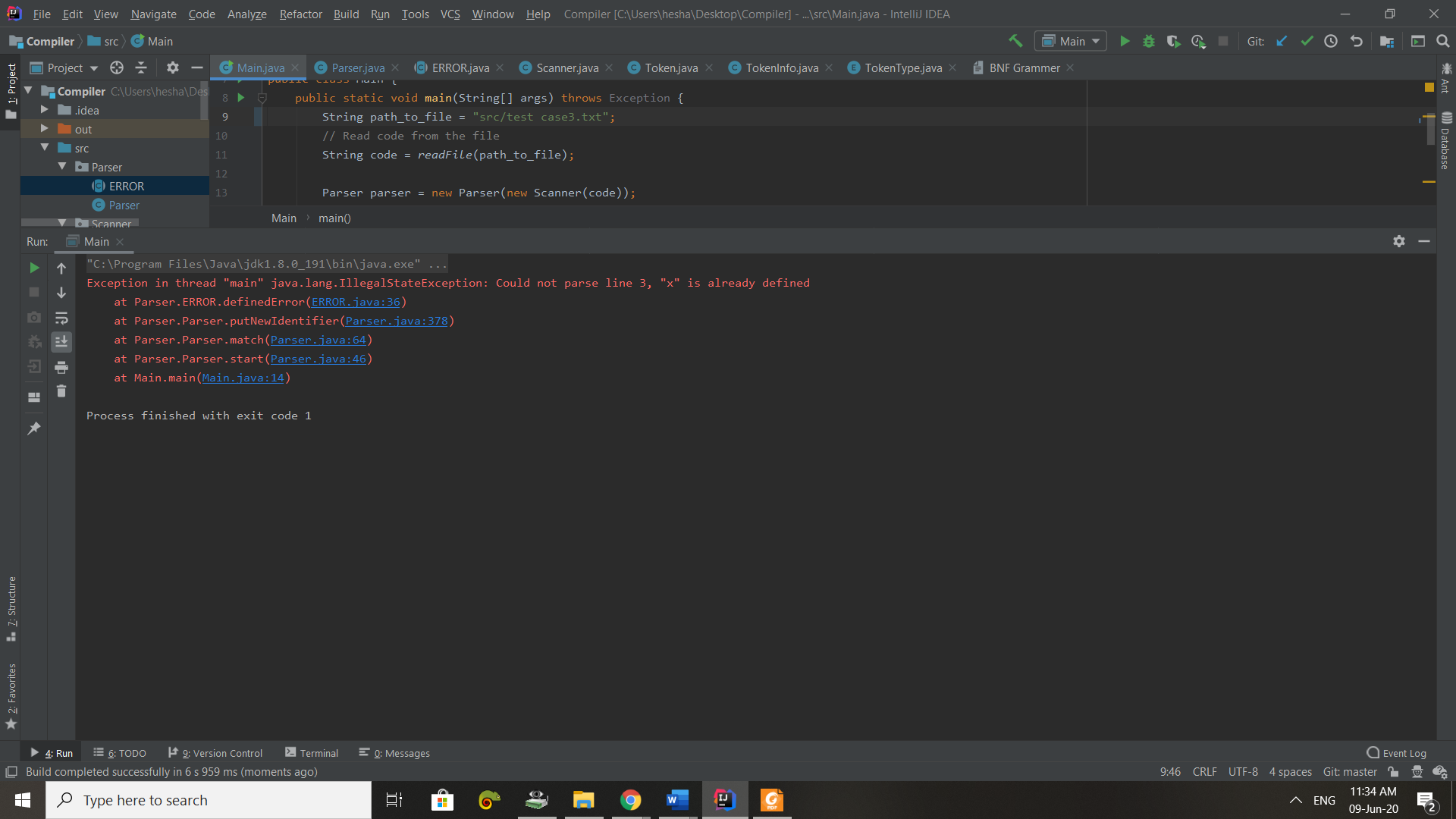
1. Test Case1



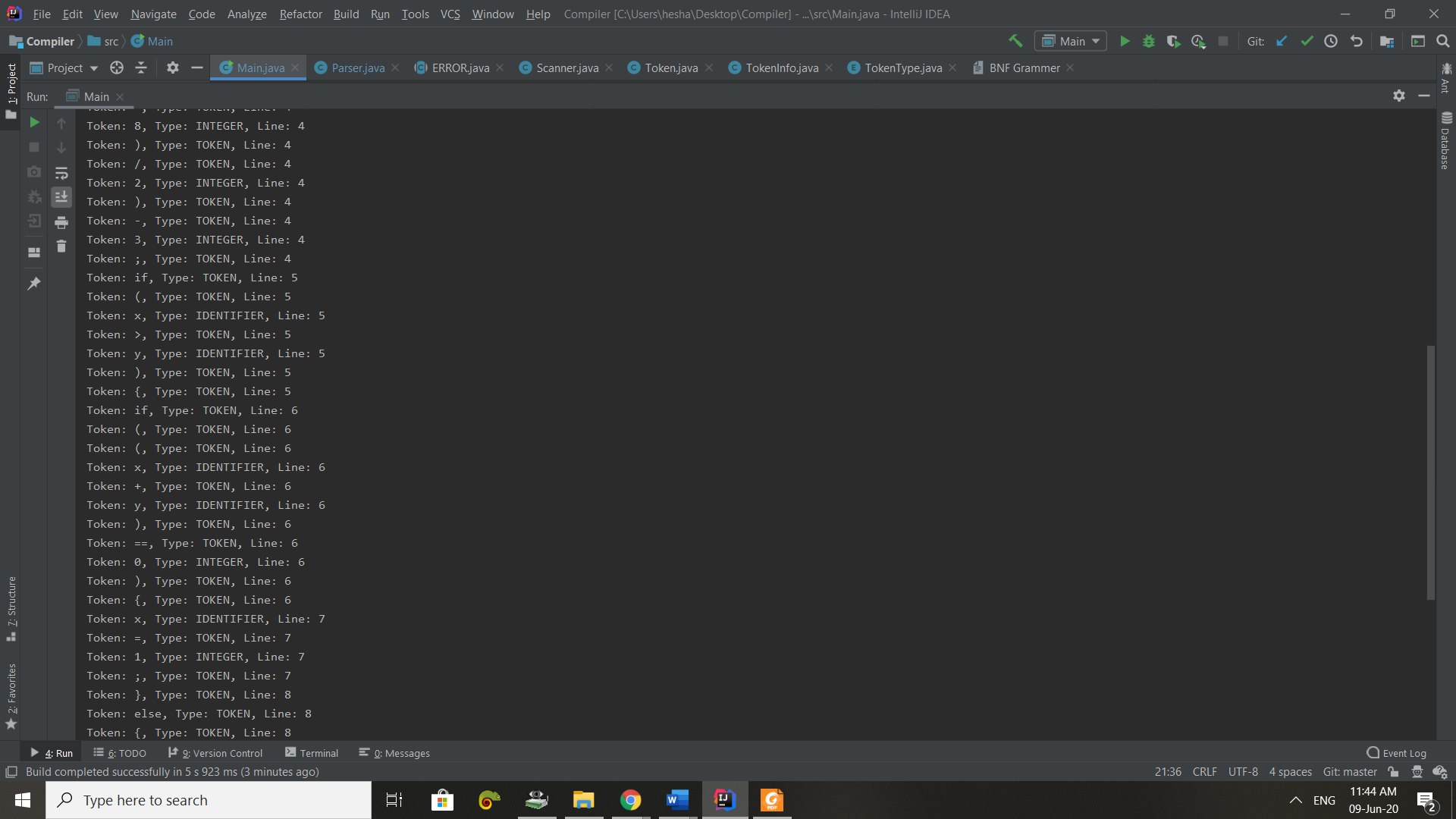
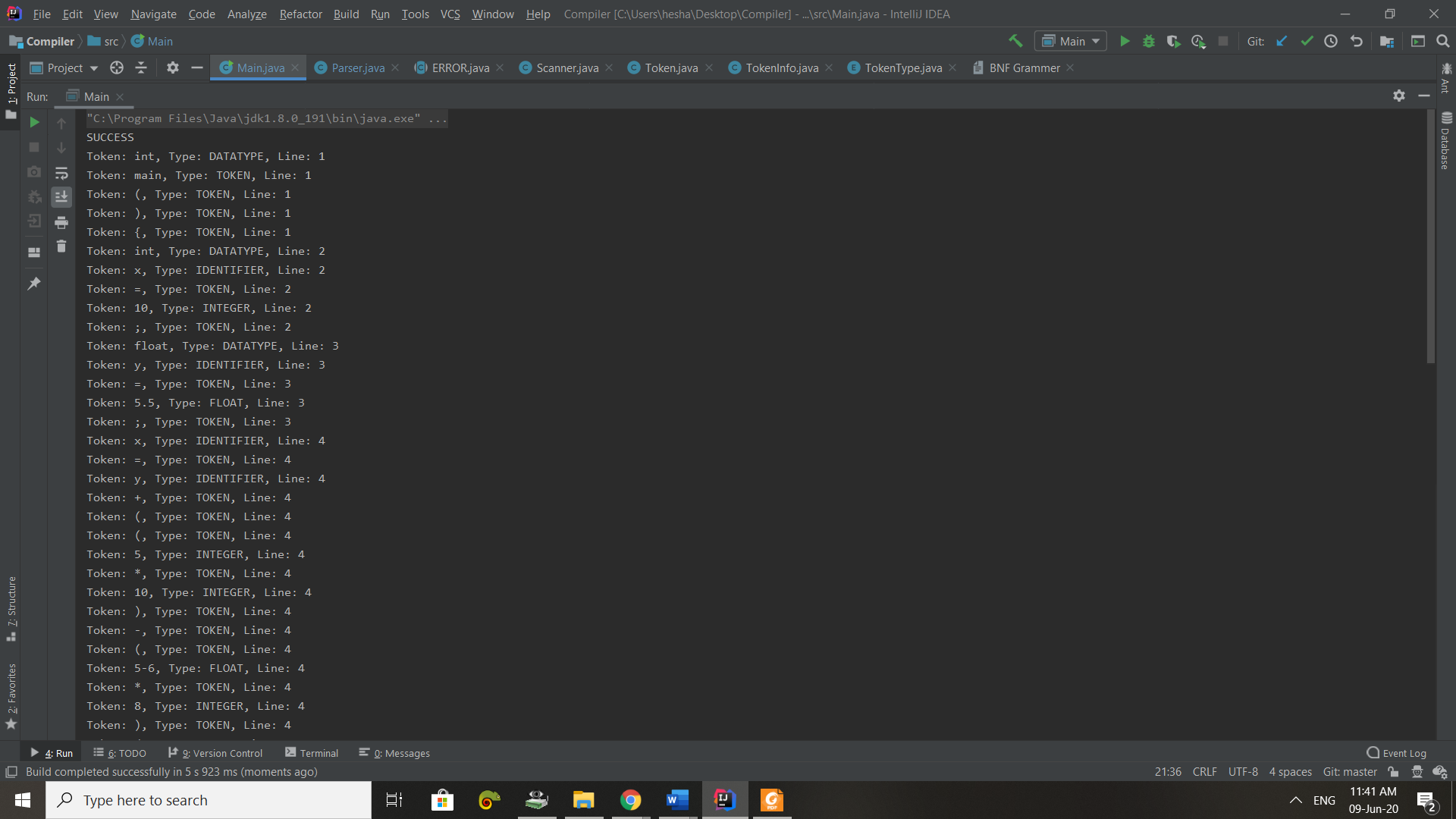
1. Test Case2

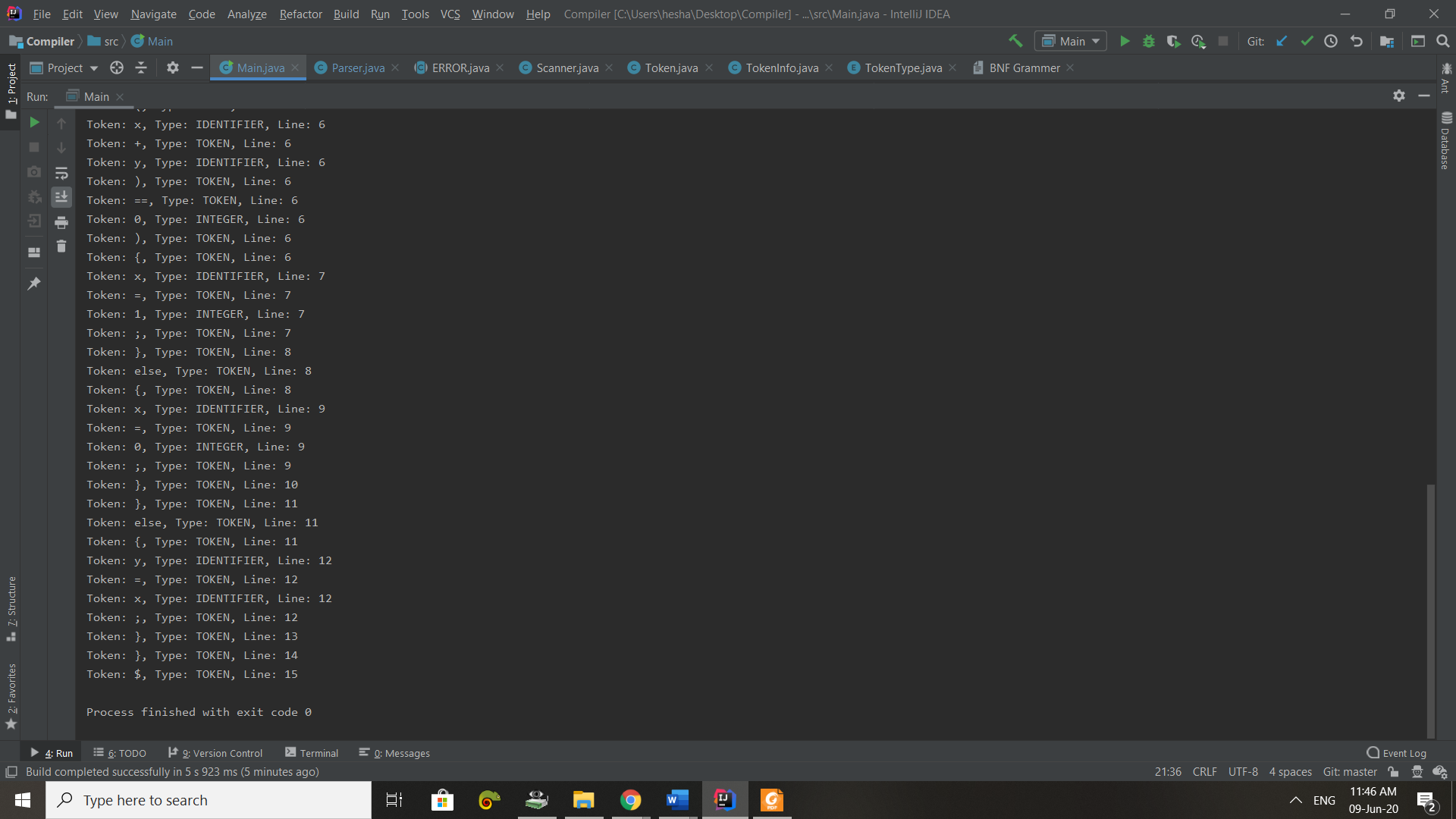


1. Test Case3

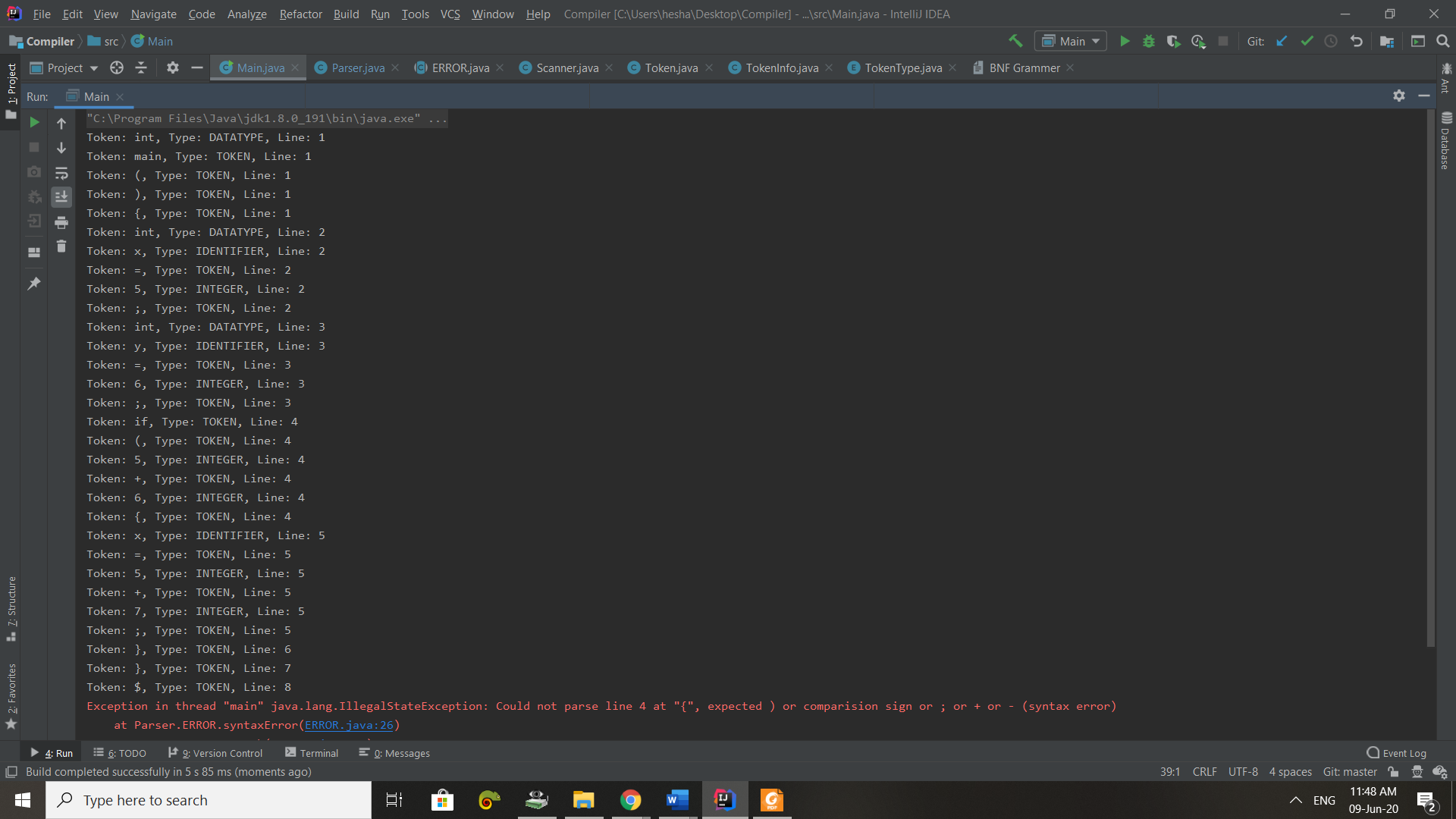


**Resulting tokens**

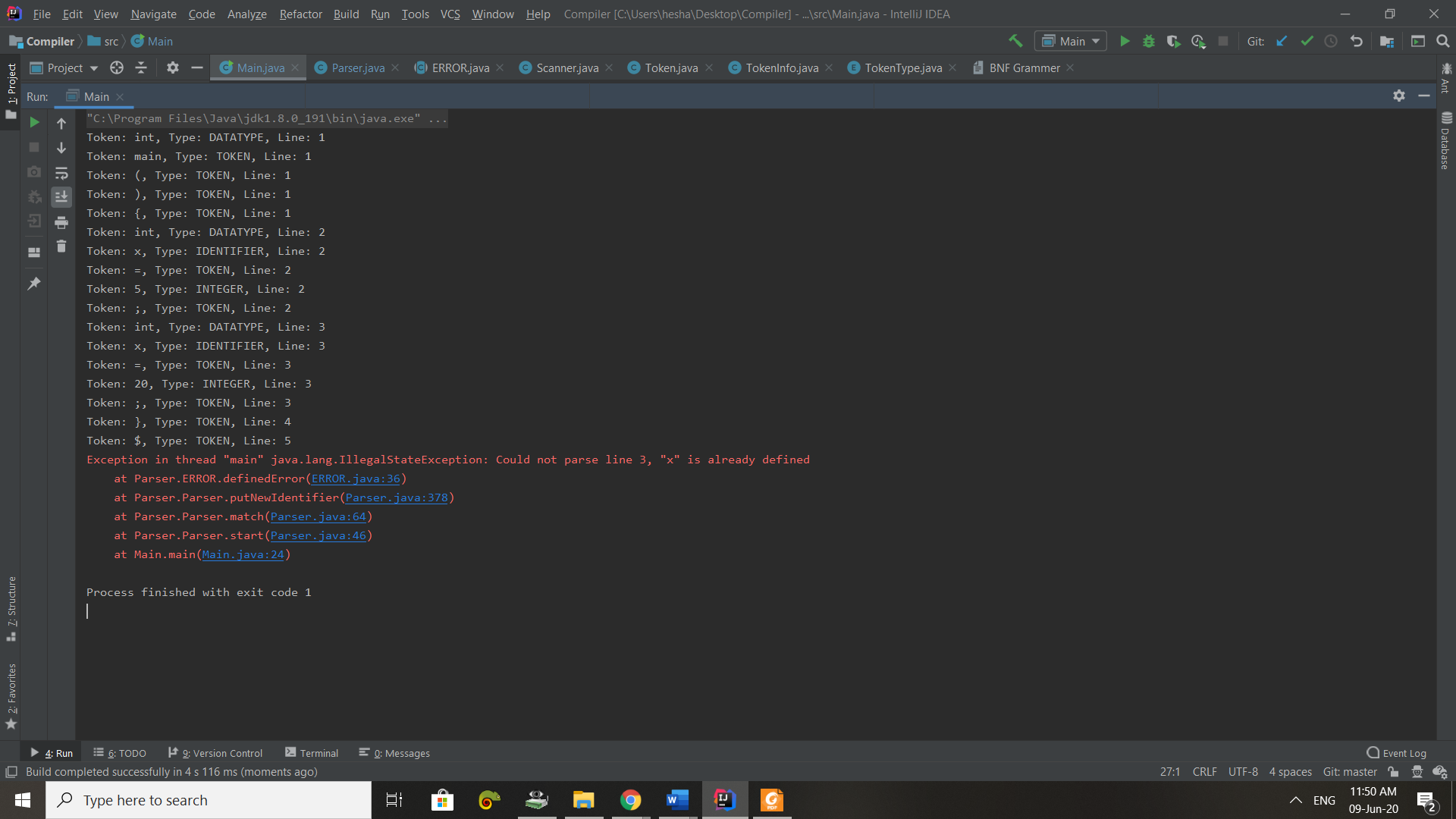
1. Test Case1



1. Test Case2



1. Test Case3



1. **Grammar**

The grammar written in BNF notation.

// note 3 = epsilon or nothing  
1- program -> stmt\_seq  
  
2- stmt\_seq -> stmt stmt\_seq'  
  
3- stmt\_seq' -> stmt\_seq | 3  
  
4- stmt -> if\_stmt | assign\_stmt ; | declare\_stmt ;  
  
5- if\_stmt -> if ( condition ) { stmt\_seq' } else\_part  
  
6- else\_part -> else { stmt\_seq' } | 3  
  
7- condition -> exp condition'  
  
8- condition' -> comp\_sign exp | 3  
  
9- comp\_sign -> < | > | == | >= | <= | !=  
  
10- exp -> term exp'  
  
11- exp' -> add\_op term exp' | 3  
  
12- add\_op -> + | -  
  
13- term -> factor term'  
  
14- term'-> mul\_op factor term' | 3  
  
15- mul\_op -> \* | /  
  
16- factor -> ( exp ) | value | id  
  
17- declare\_stmt -> datatype id x\_stmt  
  
18- x\_stmt -> = exp | 3  
  
19- assign\_stmt -> id = exp  
  
20- datatype-> int | float | char

1. **Parsing method used**

Start():

It’s the point of parser to start working, where I prepare the stack and then loop over the stack until it get empty and with every loop I call function match.

Match():

This is where the program decides what non-terminal to release or terminal to match (pop form stack and get next token) according to the current token and top of the stack, if the current token is not matching with the top then the function is using the top (string) as a name of the function to call instead of creating if-else to determine what is the appropriate function to use, where I named all the functions as names of non-terminals.

You see this line, I use string as the name of the function to call. Parser.class.getMethod(stack.pop()).invoke(this);

Then, there are 20 functions each function is releasing a specific non-terminal in the stack according to ll1 parser table and BNF grammar.

If there is a token which not matching with the current top of the stack and also after releasing all possible non-terminals then that’s an error in the code.

Epsilon();

Function to do nothing, just to simulate the parser table.

|  |  |  |
| --- | --- | --- |
| **Grammar** | **First** | **Follow** |
| 1- program -> stmt\_seq | If, id, int, float, char | $ |
| 2- stmt\_seq -> stmt stmt\_seq' | If, id, int, float, char | $,} |
| 3- stmt\_seq' -> stmt\_seq | 3 | if, id, int, float, char, 3 | $, } |
| 4- stmt -> if\_stmt | assign\_stmt ; | declare\_stmt ; | If, id, int, float, char | if, id, int, float, char, $,} |
| 5- if\_stmt -> if ( condition ) { stmt\_seq' } else\_part | If | if, id, int, float, char, $,} |
| 6- else\_part -> else { stmt\_seq' } | 3 | else, 3 | if, id, int, float, char, $,} |
| 7- condition -> exp condition' | (, value, id | ) |
| 8- condition' -> comp\_sign exp | 3 | <, >, ==, >=, <=, !=, 3 | ) |
| 9- comp\_sign -> < | > | == | >= | <= | != | <, >, ==, >=, <=, != | (, value, id |
| 10- exp -> term exp' | (, value, id | <, >, ==, >=, <=, !=, ), ; |
| 11- exp' -> add\_op term exp' | 3 | +, -, 3 | <, >, ==, >=, <=, !=, ), ; |
| 12- add\_op -> + | - | +, - | (, value, id |
| 13- term -> factor term' | (, value, id | +, -, <, >, ==, >=, <=, !=, ), ; |
| 14- term' -> mul\_op factor term' | 3 | \*, /, 3 | +, -, <, >, ==, >=, <=, !=, ), ; |
| 15- mul\_op -> \* | / | \*, / | (, value, id |
| 16- factor -> ( exp ) | value | id | (, value, id | \*, /, +, -, <, >, ==, >=, <=, !=, ), ; |
| 17- declare\_stmt -> datatype id x | Int, float, char | ; |
| 18- x\_stmt -> = exp | 3 | =, 3 | ; |
| 19- assign\_stmt -> id = exp | Id | ; |
| 20- datatype-> int | float | char | Int, float, char | id |

1. **First and Follow sets**
2. **ll1 parse table**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| line | If | Id | Int | Float | Char | Else | ( | ) | < | > | = | != | <= | >= | == | ; | value | \* | / | + | - | } | $ |
| 1- | 1 | 1 | 1 | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2- | 2 | 2 | 2 | 2 | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3- | Stmt\_seq’->stmt\_seq | Stmt\_seq’->stmt\_seq | Stmt\_seq’->stmt\_seq | Stmt\_seq’->stmt\_seq | Stmt\_seq’->stmt\_seq |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Stmt\_seq’->3 | Stmt\_seq’->3 |
| 4- | Stmt-> if\_stmt | stmt->assign\_stmt ; | Stmt->declare\_stmt ; | Stmt->declare\_stmt ; | Stmt->declare\_stmt ; |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5- | 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6- | else\_part->3 | else\_part->3 | else\_part->3 | else\_part->3 | else\_part->3 | else\_part->else{ stmt\_seq’ } |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | else\_part->3 | else\_part->3 |
| 7- |  | 7 |  |  |  |  | 7 |  |  |  |  |  |  |  |  |  | 7 |  |  |  |  |  |  |
| 8- |  |  |  |  |  |  |  | Condition’->3 | condition’->comp\_sign exp | condition’->comp\_sign exp |  | condition’->comp\_sign exp | condition’->comp\_sign exp | condition’->comp\_sign exp | condition’->comp\_sign exp |  |  |  |  |  |  |  |  |
| 9- |  |  |  |  |  |  |  |  | comp\_sign-> < | comp\_sign-> > |  | comp\_sign-> != | comp\_sign-> <= | comp\_sign-> >= | comp\_sign-> == |  |  |  |  |  |  |  |  |
| 10- |  | 10 |  |  |  |  | 10 |  |  |  |  |  |  |  |  |  | 10 |  |  |  |  |  |  |
| 11- |  |  |  |  |  |  |  | exp’-> 3 | exp’-> 3 | exp’-> 3 |  | exp’-> 3 | exp’-> 3 | exp’-> 3 | exp’-> 3 | exp’-> 3 |  |  |  | exp’->add\_op term exp’ | exp’->add\_op term exp’ |  |  |
| 12- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | add\_op-> + | add\_op-> - |  |  |
| 13- |  | 13 |  |  |  |  | 13 |  |  |  |  |  |  |  |  |  | 13 |  |  |  |  |  |  |
| 14- |  |  |  |  |  |  |  | Term’->3 | Term’->3 | Term’->3 |  | Term’->3 | Term’->3 | Term’->3 | Term’->3 | Term’->3 |  | Term->mul\_op factor term’ | Term->mul\_op factor term’ | Term’->3 | Term’->3 |  |  |
| 15- |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | mul\_op->\* | mul\_op->/ |  |  |  |  |
| 16- |  | Factor->id |  |  |  |  | Factor->( exp ) |  |  |  |  |  |  |  |  |  | Factor->value |  |  |  |  |  |  |
| 17- |  |  | 17 | 17 | 17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 18- |  |  |  |  |  |  |  |  |  |  | x\_stmt-> = exp |  |  |  |  | x\_stmt-> 3 |  |  |  |  |  |  |  |
| 19- |  | 19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  | datatype->int | Datatype->float | datatype->char |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. **Parse Tree**

Parse tree for Test Case2, but with fixing the error,

int main(){

int x = 5;

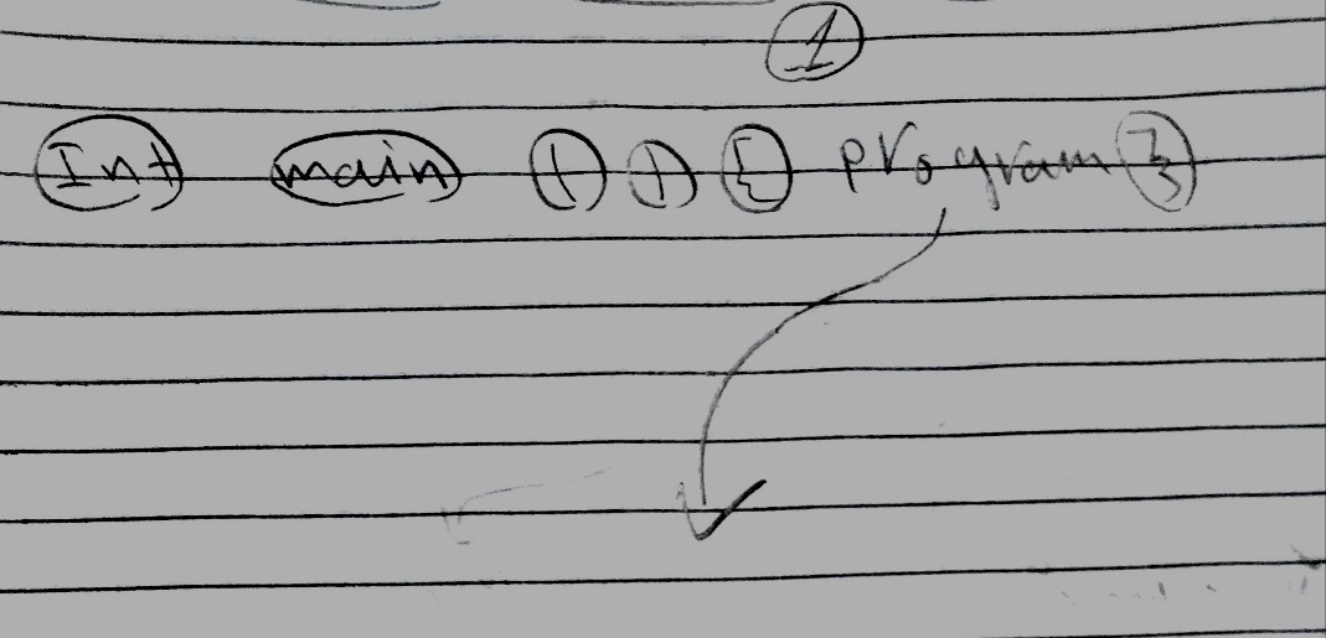
int y = 6;

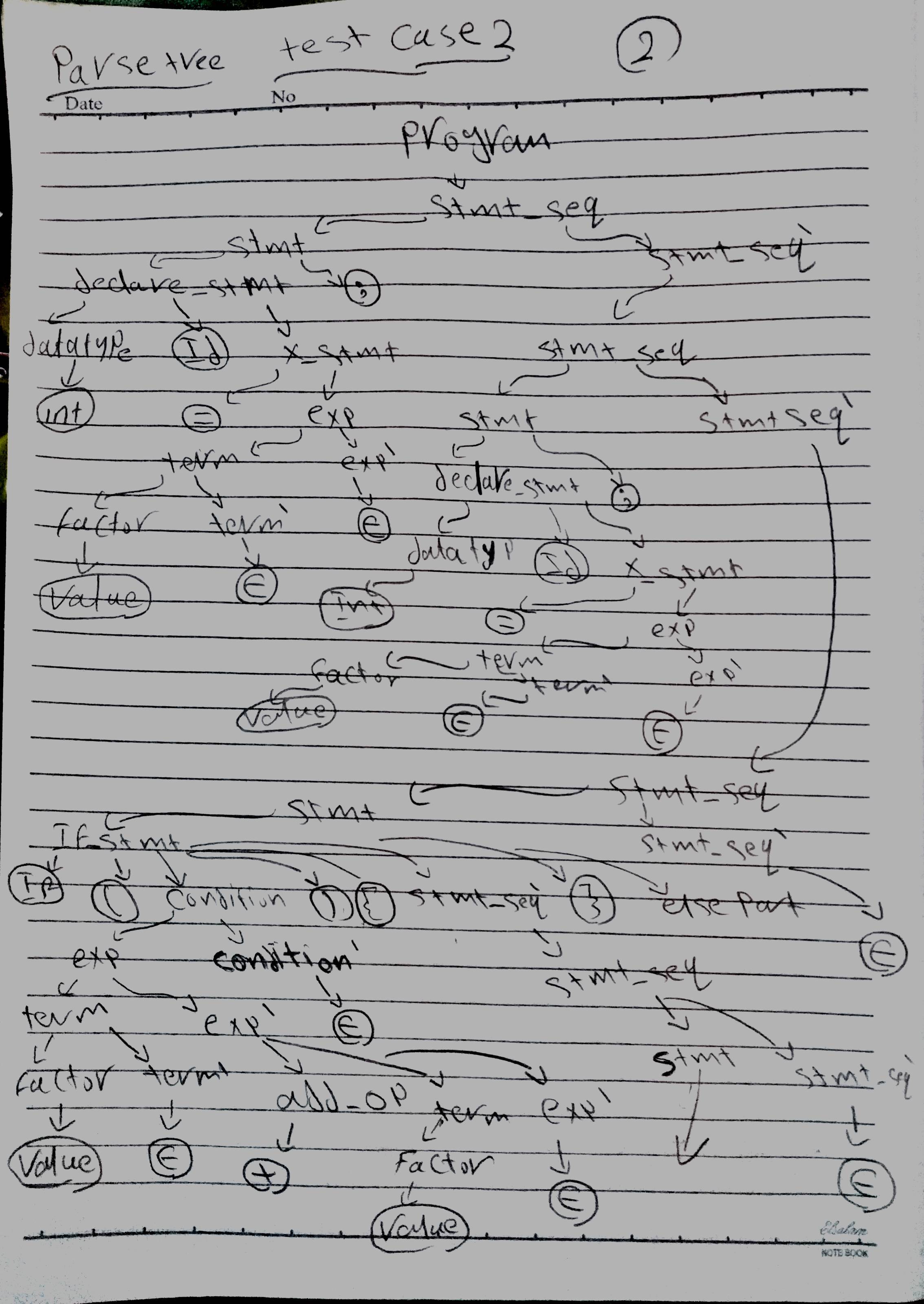
if(5 + 6){

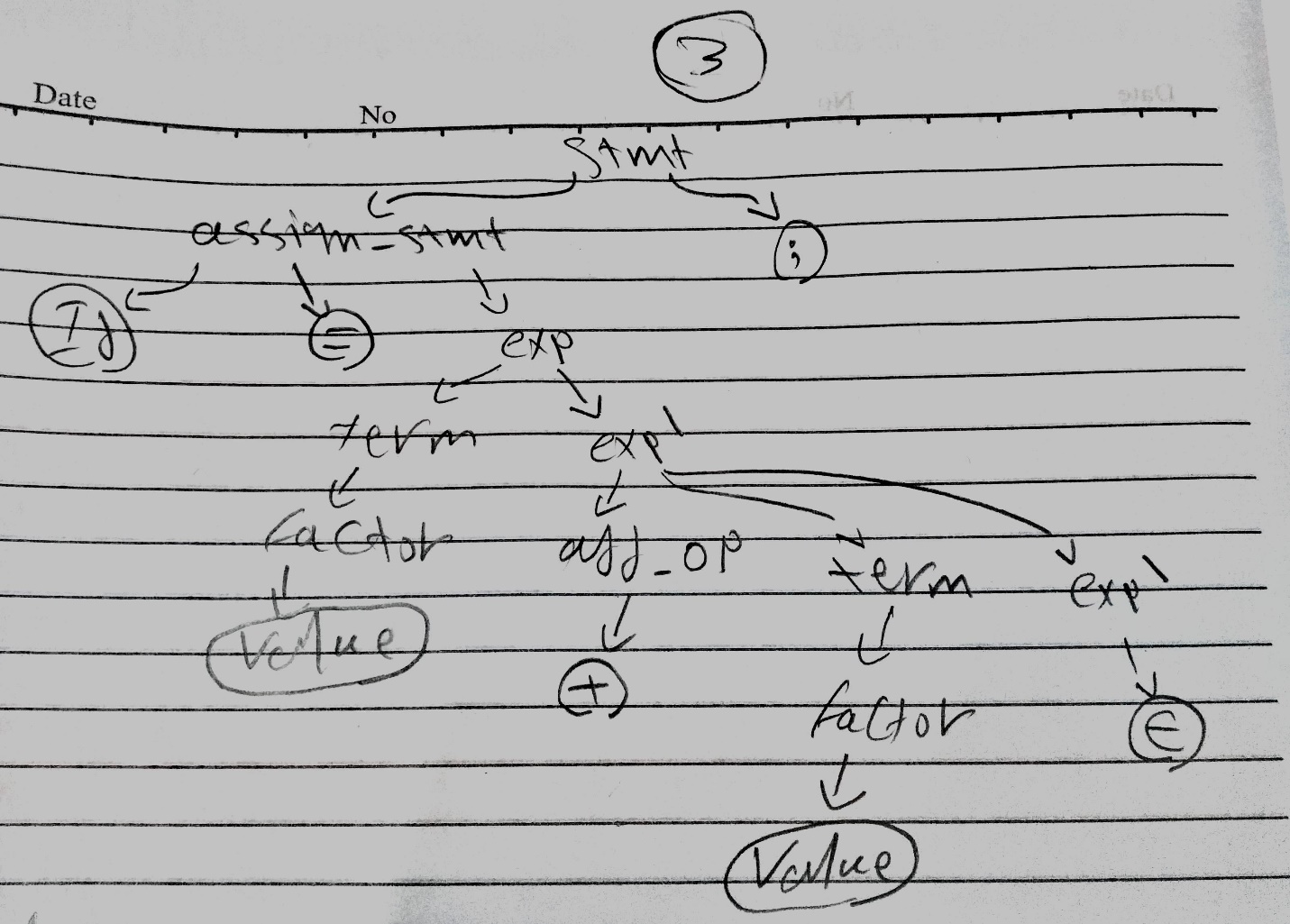
x = 5 + 7;

}

}







1. **Role of each group member**

|  |  |
| --- | --- |
| **Member** | **Task** |
| **Mahmoud Tarek Mohamed** | **Scanner** |
| **Mahmoud Salah Gad** | **Error handling and BNF grammar** |
| **Hesham Ahmed Hassan** | **Parser and ll1 parse table** |

1. **References**
2. <https://www.youtube.com/watch?v=R1ZlWEZWMKk&fbclid=IwAR3RWT3vHYTddlQn69RN60tBmoVw-qKNxrYTZdOI2Qy93aSwJoJSz6b7VYg>
3. **compiler-construction-principles-and-practice book**
4. **Slides**
5. <https://www.youtube.com/watch?v=nCiluoENyOg&list=PLQkyODvJ8ywuGxYwN0BfMSvembIJkNQH1&index=40>