Python Compiler in Java

Team Number:
5

Team Members:
Ishu Gupta: 20114041

Topic:
12. Python syntax analyzer in Java (if-else, loops, Integer, Boolean and String)

Project Description

We have built a syntax analyzer from scratch. For that, we first built a lexical analyzer that gives a stream of tokens corresponding to the given python program. Using this stream of tokens, syntax analyzer checks for the syntactical errors, if any. As mentioned in the problem statement, we have covered the following python syntaxes in our syntax analyzer-

1. Integers: This includes Integer declaration and definition (can be a simple constant or mathematical expressions like a = 10+20)
2. Boolean: This includes Boolean variable declaration and definition (can be simple assignment like a = True or simple boolean expressions like c = a<b). Boolean operators covered in our compiler are: ==,!=, <, >, <=, >=
3. String: String declaration and definition is included in this. Definition could be a simple string (like a =”hey”) or string combined with operators (like a = “abc”+”def”).

   Note: Since In compiler design, syntax analysis phase does not perform type checking (it is a part of semantic analysis), we have not checked type of operands in the expressions mentioned above. For example: a = True+10 is a valid syntax but will give error at semantic analysis phase.

4. if-else: This includes checking if, elif, else syntax. Example of a valid if, elif, else statement:

```
if a:
   b = c
elif b:
   a = c
else:
  d = c
```

   Our syntax analyzer throws an error if this syntax is violated due improper indentation, or missing (or invalid) condition in if, elif statement, getting elif/else without getting if, etc
5. for loop: This includes checking for loop syntax. Example of a valid for loop syntax:

```
for x in range(10):
   print(x)
```

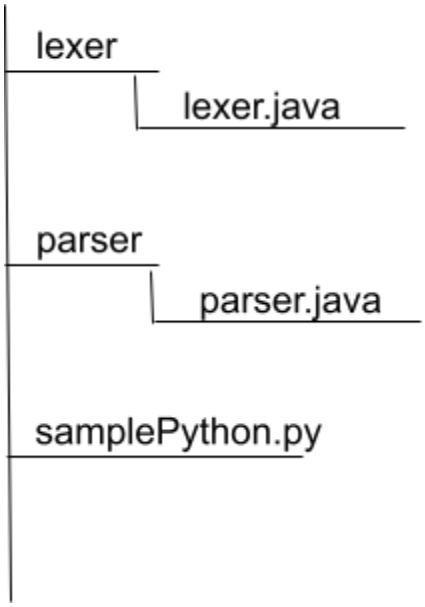
   Our syntax analyzer throws an error if this syntax is violated due to improper indentation, or missing/wrong keywords (“for”, IDENTIFIER like x, “IN”, etc).

How to Run the code?

Step 1: Place the code you want to check in samplePython.py file
Step 2: Compiler Lexical Analyzer using - javac lexer/lexer.java
Step 3: Compiler Syntax analyzer using - javac parser/parser.java
Step 4: Run parser using - java parser.parser

Code walkthrough

File Structure:



Let’s go through the code in these files:

Note: The code can also be found at - https://github.com/IshuGupta02/Python-Compiler-In-Java

lexer.java

This file contains the code for lexical analyzer.

Screenshots of the code:

```java
package lexer;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Arrays;

public class lexer{

    private static final String EMPTY_STRING = "";

    public static String ltrim(String str) {
        return str.replaceAll("^\\s+", EMPTY_STRING);
    }

    public static String rtrim(String str) {
        return str.replaceAll("\\s+$", EMPTY_STRING);
    }

    public static boolean checkInteger(String token, ArrayList<String> tokens){
        for(int i=0; i<token.length(); i++){
            if(!(token.charAt(i)>='0' && token.charAt(i)<='9')){
                return false;
            }
        }

        tokens.add("CONST_INTEGER");
        return true;

    }
```

lexer class is the class that contains all the methods and variables for lexical analysis.

Function Descriptions:

1. ltrim, rtrim is used to remove leading and trailing spaces respectively
2. checkInteger function checks whether the given string is an Integer constant or not

```java
public static boolean checkBoolean(String token, ArrayList<String> tokens){
    if(token.equals("False")){
        tokens.add("CONST_FALSE");
        return true;

    }
    else if(token.equals("True")){
        tokens.add("CONST_TRUE");
        return true;

    }

    return false;

}
public static boolean checkString(String token, ArrayList<String> tokens){
    if((token.charAt(0)=='"' && token.charAt(token.length()-1)=='"')){
        for(int i=1; i<token.length()-1; i++){
            if(token.charAt(i)=='"'){
                return false;
            }
        }

        tokens.add("CONST_STRING");

        return true;

    }

    if((token.charAt(0)=='\'' && token.charAt(token.length()-1)=='\'')){
        for(int i=1; i<token.length()-1; i++){
            if(token.charAt(i)=='\''){
                return false;
            }
        }

        tokens.add("CONST_STRING");

        return true;

    }

    return false;

}
```

Function Descriptions:

1. checkBoolean function checks whether the given string is a boolean constant or not
2. checkString function checks whether the given string is a valid string constant or not

```
79      public static boolean checkIdentifier(String token, ArrayList<String> tokens){
80
81          if(token.charAt(0)>='0' && token.charAt(0)<='9'){
82              return false;
83          }
84          for(int i=1; i<token.length(); i++){
85              if(!(
86                  (token.charAt(i)>='a' && token.charAt(i)<='z')||
87                  (token.charAt(i)>='A' && token.charAt(i)<='Z')||
88                  (token.charAt(i)>='0' && token.charAt(i)<='9')||
89                  (token.charAt(i)=='_')
90
91              )){
92                  return false;
93
94              }
95          }
96
97          tokens.add("IDENTIFIER");
98          return true;
99
100     }
```

checkIdentifier functions checks whether the given string is a valid python identifier or not. Identifiers in Python can comprise both lowercase (a to z) and uppercase letters (A to Z). They can also include digits (0 to 9) and start with an underscore (_)

```java
public static boolean checkOperator(String token, ArrayList<String> tokens){
    // ':', '=', '+', '-', '/', '*', '==', '(', ')', '[', ']', '{', '}'
    if(token.equals(":")){
        tokens.add("COLON");
        return true;
    }
    else if(token.equals("=")){
        tokens.add("EQUALS");
        return true;
    }
    else if(token.equals("+")){
        tokens.add("PLUS");
        return true;
    }
    else if(token.equals("-")){
        tokens.add("MINUS");
        return true;
    }
    else if(token.equals("/")){
        tokens.add("SLASH");
        return true;
    }
    else if(token.equals("*")){
        tokens.add("STAR");
        return true;
    }
    else if(token.equals("==")){
        tokens.add("DOUBLE_EQUALS");
        return true;
    }
    else if(token.equals("(")){
        tokens.add("OPEN_PAREN");
        return true;
    }
    else if(token.equals(")")){
        tokens.add("CLOSE_PAREN");
```

This is a part of checkOperator functions which checks for operators like +, -, /, *, etc

```
174     public static boolean checkKeyword(String token, ArrayList<String> tokens){
175
176         if(token.equals("if")){
177             tokens.add("IF");
178             return true;
179
180         }
181         else if(token.equals("else")){
182             tokens.add("ELSE");
183             return true;
184
185         }
186         else if(token.equals("elif")){
187             tokens.add("ELIF");
188             return true;
189
190         }
191         else if(token.equals("for")){
192             tokens.add("FOR");
193             return true;
194
195         }
196         else if(token.equals("pass")){
197             tokens.add("PASS");
198             return true;
199         }
200         else if(token.equals("in")){
201             tokens.add("IN");
202             return true;
203         }
204
205
206         return false;
207
208     }
```

checkKeyword functions add keywords from the program to out stream of tokens

```
    public static boolean fillToken(String tokenFound, ArrayList<String> tokens){
        if(!checkKeyword(tokenFound, tokens)){
            if(!checkInteger(tokenFound, tokens)){
                if(!checkBoolean(tokenFound, tokens)){
                    if(!checkString(tokenFound, tokens)){
                        if(!checkIdentifier(tokenFound, tokens)){
                            return false;

                        }
                    }
                }
            }

        }

        return true;

    }
```

fillToken function calls the functions explained above. It returns true if the given string matches with one of the tokens, else returns false.

```
    public static void checkToken(String token, ArrayList<String> tokens){

        int lastToken = -1;

        for(int i=0; i<token.length(); i++){
            if(Arrays.asList(':', '=', '+', '-', '/', '*', '(', ')', '[', ']', '{', '}', '>', '<', '!').contains(token.charAt(i))){
                String tokenFound = token.substring(lastToken+1,i);

                if(lastToken<i-1 && !fillToken(tokenFound, tokens)){
                    // error
                    System.out.println("Not a valid token: "+tokenFound);

                }
                if(!checkOperator(token.charAt(i)+"", tokens)){
                    //error
                    System.out.println("Not a valid token: "+tokenFound);
                }

                lastToken = i;

            }
            else{

            }
        }

        if(lastToken!=token.length()-1){
            String token_left = token.substring(lastToken+1,token.length());

            if(!fillToken(token_left, tokens)){
                // error
                System.out.println("Not a valid token: "+token_left);
            }
        }


    }
```

```java
public static ArrayList<String> tokenize(){

    ArrayList<String> tokens = new ArrayList<>();
    try {
        File myObj = new File("samplePython.py");
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            data = rtrim(data);

            int spaceCount = 0;
            int lastToken = -1;

            for(int i=0; i<data.length(); i++){
                if(data.charAt(i)==' '){
                    if(spaceCount == 0 && lastToken+1<i){
                        checkToken(data.substring(lastToken+1, i), tokens);
                    }

                    spaceCount++;
                    lastToken = i;

                    if(spaceCount == 4){
                        tokens.add("TAB");
                        spaceCount = 0;
                    }
                }
                else{

                    for(int j=0; j<spaceCount; j++){
                        tokens.add("SPACE");
                        lastToken = i-1;
                    }

                    spaceCount = 0;
                }
            }

            if(lastToken!=data.length()-1){
                checkToken(data.substring(lastToken+1,data.length()), tokens);
            }


            if(myReader.hasNextLine()) tokens.add("NEWLINE");
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println(e);
        e.printStackTrace();
    }

    return tokens;

}
```

parser.java

This file contains the code for syntax analyzer.

Screenshots of the code:

```java
package parser;

import lexer.lexer;

import java.util.*;

public class parser{

    public static int expected_tabs = 0;
    public static boolean exact_tabs_required = true;
    public static HashSet<Integer> if_indents = new HashSet<Integer>();

    public static ArrayList<ArrayList<String>> repaceEqualDoubleEquals(ArrayList<ArrayList<String>> tokens){

        ArrayList<ArrayList<String>> answer = new ArrayList<>();

        for(int i=0; i<tokens.size(); i++){
            ArrayList<String> line = tokens.get(i);

            ArrayList<String> line_formated = new ArrayList<>();

            for(int j=0; j<line.size(); j++){
                if(j<line.size()-1 && line.get(j).equals("EQUALS") && line.get(j+1).equals("EQUALS")){
                    line_formated.add("DOUBLE_EQUALS");
                    j++;
```

parser class is the class that contains all the methods and variables for syntactic analysis.

```java
public static ArrayList<ArrayList<String>> seperateLines(ArrayList<String> tokens){
    ArrayList<ArrayList<String>> new_list = new ArrayList();
    if(tokens.size()==0){
        return new_list;
    }

    ArrayList<String> list_= new ArrayList();
    for(int i=0; i<tokens.size(); i++){
        if(tokens.get(i).equals("NEWLINE")){
            new_list.add(list_);
            list_ = new ArrayList();
        }
        else{
            list_.add(tokens.get(i));
        }
    }

    new_list.add(list_);

    return new_list;

}
```

seperateLines method is a helpful function which uses newline character as a delimiter to separate the stream of tokens with the help of newline.

```java
public static ArrayList<Integer> removeIntermediateSpaces(ArrayList<ArrayList<String>> tokens) {

    ArrayList<Integer> tabCount = new ArrayList();

    for(int i=0; i<tokens.size(); i++){
        boolean nonSpaceFound = false;

        ArrayList<String> old_list = tokens.get(i);
        ArrayList<String> new_list = new ArrayList();

        int tabs = 0;

        for(int j=0; j<old_list.size(); j++){
            if(old_list.get(j).equals("TAB")){
                if(!nonSpaceFound){
                    tabs++;
                }
            }
            else if(old_list.get(j).equals("SPACE")){
                if(!nonSpaceFound){
                    throw new RuntimeException("Unexpected Indent at Line number: "+i);
                }
            }
            else{
                nonSpaceFound = true;
                new_list.add(old_list.get(j));
            }
        }

        tabCount.add(tabs);
        tokens.set(i, new_list);

    }

    return tabCount;
}
```

removeIntermediate spaces is a helper function that removes extra (unwanted) spaces or tabs from the stream of tokens. Additionally, it checks for the number of spaces in the beginning of any line and notes it down in an array called tabCount.
(tabCount is quite useful in syntax analysis of python as in python indentation plays an important role)

```
130     public static boolean checkIfElse(ArrayList<String> tokens, int tab_count){
131
132         if(tokens.get(0)=="IF"){
133             try{
134                 if(!checkExpression(tokens, 1, tokens.size()-2)){
135                     throw new RuntimeException("Invalid ELIF condition");
136                 }
137                 if(!tokens.get(tokens.size()-1).equals("COLON")){
138                     throw new RuntimeException("Invalid End of line in ELIF");
139                 }
140                 if_indents.add(tab_count);
141
142             }
143             catch(Exception e){
144                 throw new RuntimeException("Invalid IF statement");
145             }
146
147         }
148         else if(tokens.get(0)=="ELIF"){
149             try{
150                 if(!if_indents.contains(tab_count)){
151                     throw new RuntimeException("ELIF without IF found");
152                 }
153                 else{
154                     if(!checkExpression(tokens, 1, tokens.size()-2)){
155                         throw new RuntimeException("Invalid ELIF condition");
156                     }
157                     if(!tokens.get(tokens.size()-1).equals("COLON")){
158                         throw new RuntimeException("Invalid End of line in ELIF");
159                     }
160                 }
161
162             }
163             catch(Exception e){
164                 throw new RuntimeException("Invalid ELIF statement");
165             }
166
167         }
```

```
168         else if(tokens.get(0)=="ELSE"){
169
170             try{
171                 if(!if_indents.contains(tab_count)){
172                     throw new RuntimeException("ELSE without IF found");
173                 }
174                 else{
175                     removeIfIndentsGreaterThanEqualTo(tab_count);
176                 }
177
178                 if(tokens.get(1).equals("COLON")){
179                     exact_tabs_required= true;
180                     expected_tabs = tab_count+1;
181                 }
182                 else{
183                     throw new RuntimeException("Invalid Else statement");
184                 }
185
186             }
187             catch(Exception e){
188                 throw new RuntimeException("Invalid Else statement");
189             }
190
191         }
192         else{
193             return false;
194         }
195
196         return true;
197
198     }
```

checkIfElse function checks if the given line is a valid if else statement.

```
200     public static boolean checkForLoop(ArrayList<String> tokens, int tab_count){
201         // System.out.println(tokens);
202         try{
203             if(tokens.get(0).equals("FOR")){
204                 if(tokens.get(1).equals("IDENTIFIER")){
205                     if(tokens.get(2).equals("IN")){
206                         int i=3;
207                         while(!tokens.get(i).equals("COLON")){
208                             i++;
209                         }
210 
211                         if(!tokens.get(i).equals("COLON") || i!=tokens.size()-1){
212 
213                             // System.out.println("no1");
214                             throw new RuntimeException("Invalid for loop");
215                         }
216 
217                         expected_tabs = tab_count+1;
218                         exact_tabs_required = true;
219 
220                         return true;
221 
222                     }
223                     else{
224                         // System.out.println("no2");
225                         throw new RuntimeException("Invalid for loop");
226                     }
227                 }
228                 else{
229                     // System.out.println("no3");
230                     throw new RuntimeException("Invalid for loop");
231                 }
232             }
233             else{
234                 return false;
235             }
236 
237         }
238         catch(Exception e){
239             return false;
240         }
241     }
```

checkForLoop function checks if the given line is a valid for statement.

```
362     public static boolean checkAssignment(ArrayList<String> tokens, int tab_count){
363
364         try{
365             if(tokens.get(0).equals("IDENTIFIER")){
366                 if(tokens.get(1).equals("EQUALS")){
367                     if(checkExpression(tokens, 2, tokens.size()-1)){
368                         exact_tabs_required = false;
369                         expected_tabs = tab_count;
370                         return true;
371                     }
372                     else{
373                         return false;
374                     }
375                 }
376                 else{
377                     return false;
378                 }
379
380             }
381             else{
382                 return false;
383             }
384
385         }
386         catch(Exception e){
387             return false;
388
389         }
390
391     }
392
393     public static boolean checkStatement(ArrayList<String> tokens, int tab_count){
394         if(checkAssignment(tokens, tab_count)){
395             expected_tabs = tab_count;
396             exact_tabs_required = false;
397             return true;
398         }
399
400         return false;
401     }
402
```

These two functions check for valid statements which in turn checks for valid assignments.

```java
public static boolean checkExpression(ArrayList<String> tokens, int first_token, int last_token){
    boolean result = true;
    Stack<String> st1 = new Stack<>();
    Stack<String> st2 = new Stack<>();
    boolean isTrue = true;
    for (int i = first_token; i <= last_token; i++) {
        String temp = tokens.get(i);
        if (isDigit(temp)) {
            st1.push(temp);
            if(isTrue) {
                isTrue = false;
            }
            else {
                return false;
            }
        }
        else if (isOperator(temp)) {
            st2.push(temp);
            isTrue = true;
        }
        else {
            if(isBracketOpen(temp)) {
                st2.push(temp);
            }
            else {
                boolean flag = true;
                while (!st2.isEmpty()) {
                    String c = st2.pop();
                    if (c.equals(getCorrespondingChar(temp))) {
                        flag = false;
                        break;
                    }
                    else {
                        if (st1.size() < 2) {
                            return false;
                        }
                        else {
                            st1.pop();
                        }
                    }
                }
                if (flag) {
                    return false;
                }
```

```java
        while (!st2.isEmpty()) {
            String c = st2.pop();
            if (!isOperator(c)) {
                return false;
            }
            if (st1.size() < 2) {
                return false;
            }
            else {
                st1.pop();
            }
        }
        if (st1.size() > 1 || !st2.isEmpty()) {
            return false;
        }
        return result;
    }
```

This part of code checks for valid expressions like - a+b*c/d

```java
public static String getCorrespondingChar(String c) {
    if (c.equals("OPEN_PAREN")) {
        return "CLOSE_PAREN";
    }
    else if (c.equals("OPEN_BRAC")) {
        return "CLOSE_BRAC";
    }
    return "CLOSE_CURLY";
}

public static boolean isBracketOpen(String c) {
    if (c.equals("OPEN_PAREN") || c.equals("OPEN_BRAC")|| c.equals("OPEN_CURLY")) {
        return true;
    }
    return false;
}

public static boolean isDigit(String c) {
    if (c.equals("CONST_INTEGER") ||
    c.equals("CONST_FALSE") ||
    c.equals("CONST_TRUE") ||
    c.equals("CONST_STRING") ||
    c.equals("IDENTIFIER")) {
        return true;
    }
    return false;
}

public static boolean isOperator(String c) {
    if (c.equals("PLUS") ||
    c.equals("MINUS") ||
    c.equals("STAR") ||
    c.equals("SLASH") ||
    c.equals("DOUBLE_EQUALS") ||
    c.equals("NOT_EQUALS") ||
    c.equals("LESS_THAN") ||
    c.equals("GREATER_THAN") ||
    c.equals("LESS_THAN_EQUALS") ||
    c.equals("GREATER_THAN_EQUALS")
    ) {
        return true;
    }
    return false;
}
```

These are the helper functions for checking expressions.

```java
public static void checkSyntax(ArrayList<ArrayList<String>> line_seperated_tokens, ArrayList<Integer> tab_count){

    boolean noError = true;
    for(int i=0; i<line_seperated_tokens.size(); i++){

        try{
            ArrayList<String> tokens = line_seperated_tokens.get(i);
            if(tokens.size()==0) continue;

            if(exact_tabs_required){
                if(tab_count.get(i)!=expected_tabs){
                    throw new RuntimeException("Unexpected Indent at Line number: "+(i+1));
                }
            }
            else{
                if(tab_count.get(i)>expected_tabs){
                    throw new RuntimeException("Unexpected Indent at Line number: "+(i+1));
                }

            }
            if(!checkIfElse(tokens, tab_count.get(i))){
                if(!checkForLoop(tokens, tab_count.get(i))){
                    if(!checkStatement(tokens, tab_count.get(i))){
                        throw new RuntimeException("Not a valid construct at: "+(i+1));
                    }
                }
                removeIfIndentsGreaterThanEqualTo(tab_count.get(i));
            }
            else{
                removeIfIndentsGreaterThan(tab_count.get(i));
            }
        }
        catch(Exception e){
            noError = false;
            System.out.println("Line: "+(i+1)+" : "+ e);
        }
    }

    if(noError){
        System.out.println("All okay in syntax!");
    }
}

```

This function checks for syntax errors by checking if the syntax matches with any of the valid syntaxes like if-else, for loop, etc

Test cases

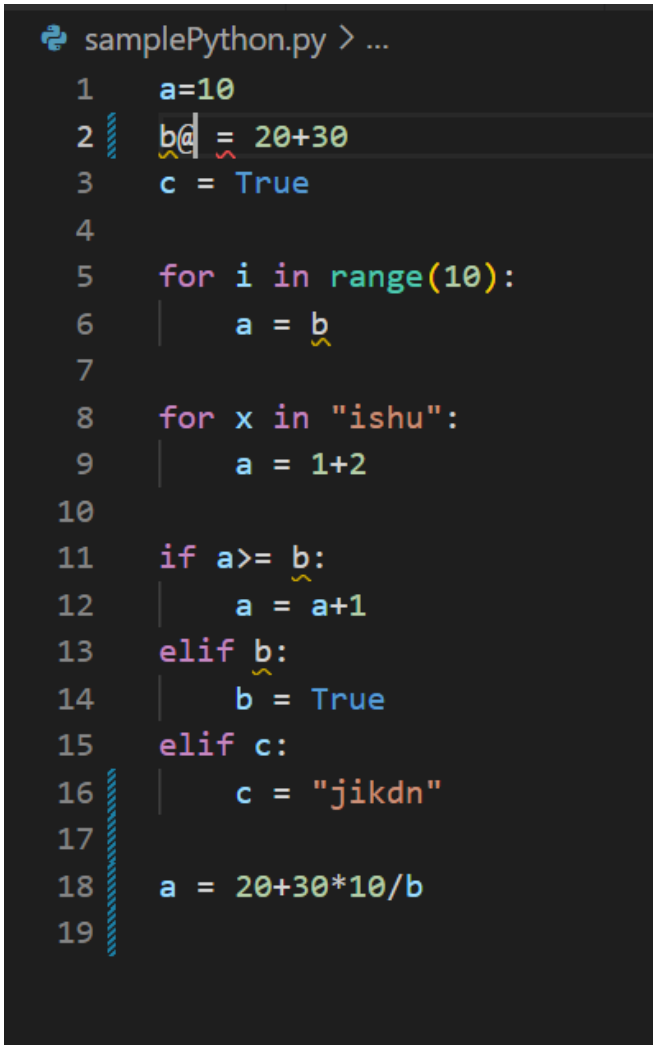
samplePython.py > ...

```python
a=10
b = 20+30
c = True

for i in range(10):
    a = b

for x in "ishu":
    a = 1+2

if a>= b:
    a = a+1
elif b:
    b = True
elif c:
    c = "jikdn"

a = 20+30*10/b

```

```
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ javac lexer/lexer.java
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ javac parser/parser.java
Note: parser/parser.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ java parser.parser
All okay in syntax!
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$
```

Since, there is no syntactical error in the sample code, our syntax analyzer says “All okay in syntax!”

Note the change in line number 2, this should come out as lexical error since b@ is not any valid token




```
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ java parser.parser
Not a valid token: b@
Unexpected Indent at Line number: 2
Line: 2 : java.lang.RuntimeException: Not a valid construct at: 2
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$
```

First, it encounters a lexical error as b@ is not a valid token
Then, since b@ has been ignored in later phases of compiler, it throws Indentation error as there will be an extra space left
And lastly, in the same line since removal of b@ will render this line syntactically wrong, it throws an error

Note the change in Line number 5 and 9

samplePython.py > ...

```
1   a=10
2   b = 20+30
3   c = True
4
5   far i in range(10):
6       a = b
7
8   for x in "ishu":
9   a = 1+2
10
11  if a>= b:
12      a = a+1
13  elif b:
14      b = True
15  elif c:
16      c = "jikdn"
17
18  a = 20+30*10/b
19
```

```
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ java parser.parser
Line: 5 : java.lang.RuntimeException: Not a valid construct at: 5
Line: 6 : java.lang.RuntimeException: Unexpected Indent at Line number: 6
Line: 9 : java.lang.RuntimeException: Unexpected Indent at Line number: 9
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$
```

Line 5 has “far” instead of “for”, and hence the error
Since there is no valid for loop at line 5, line 6 should be aligned with indentation = 0
Also, Line 9 should have one tab space more than what it has right now, hence we got an error.

Note the change in expressions in Line number 2 and 4

samplePython.py > ...

```
1   a=10
2   b = 20+30+a-
3   c = 10<2
4   d = (10+20)
5
6   for i in range(10):
7       a = b
8
9   for x in "ishu":
10      a = 1+2
11
12  if a>= b:
13      a = a+1
14  elif b:
15      b = True
16  elif c:
17      c = "jikdn"
18
19  a = 20+30*10/b
20
```

```
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ java parser.parser
Line: 2 : java.lang.RuntimeException: Not a valid construct at: 2
Line: 4 : java.lang.RuntimeException: Not a valid construct at: 4
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$
```

Since, these are not valid expressions, we get syntax error in these 2 lines

samplePython.py > ...

```python
a = 1
b = 20+30
c = 10<2
d = 10+1

for i in range(10):
    a = b

for x in "ishu":
    a = 1+2

if :
    a = a+1
elif :
    b = True
elif :
    c = "jikdn"

a = 20+30*10/b

```

```
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$ java parser.parser
Line: 12 : java.lang.RuntimeException: Invalid IF statement
Line: 14 : java.lang.RuntimeException: Invalid ELIF statement
Line: 16 : java.lang.RuntimeException: Invalid ELIF statement
ishu@DESKTOP-H7OUEF9:/mnt/e/semester 6/compiler/cp1$
```

If, elif statements in line number 12, 14, 16 is invalid