

Project Report

CSN-352

Compiler Design

Group ID: 29

Member 1: Rajat Raj Singh E. No: 21114079

Member 2: Pranavdeep Singh E. No: 21119036

Member 3: Piyush Arya E. No: 21114074

Project ID: 3

Problem Statement: Implementing a lexer and parser with LLVM compilation framework to compile Python codes and keep track of basic constructs like loops, conditional branching, function, etc, while also checking for any lexing and parsing errors.

Implementation details:

Our project implementation includes 2 files- Lexer.cpp and Parser.cpp.

The lexer file handles the linear analysis of the source file to output the stream of tokens. The stream of tokens is then used by the parser file to generate the parse tree of the source code.

The lexer works as follows:

The lexer takes the source code from the sourceCode.py file, which contains the input Python code. The details of writing the Python code are specified in the readme file. The lexer first preprocesses the source file to replace the Python indentation notation with the corresponding bracket notation({...}) to simplify the parsing stage. The preprocessed code is stored in the PreprocessedOutput.txt file, which is read one character at a time and is matched with the Regular expressions of various tokens. The regular expression matching is implemented using the DFA class, which follows the principles of Deterministic Finite Automata. All the lexemes and their corresponding tokens are written into the LexerOutput.txt file and if there is any lexical error, it is reported along with the matched tokens till the error is encountered.

The parser works as follows:

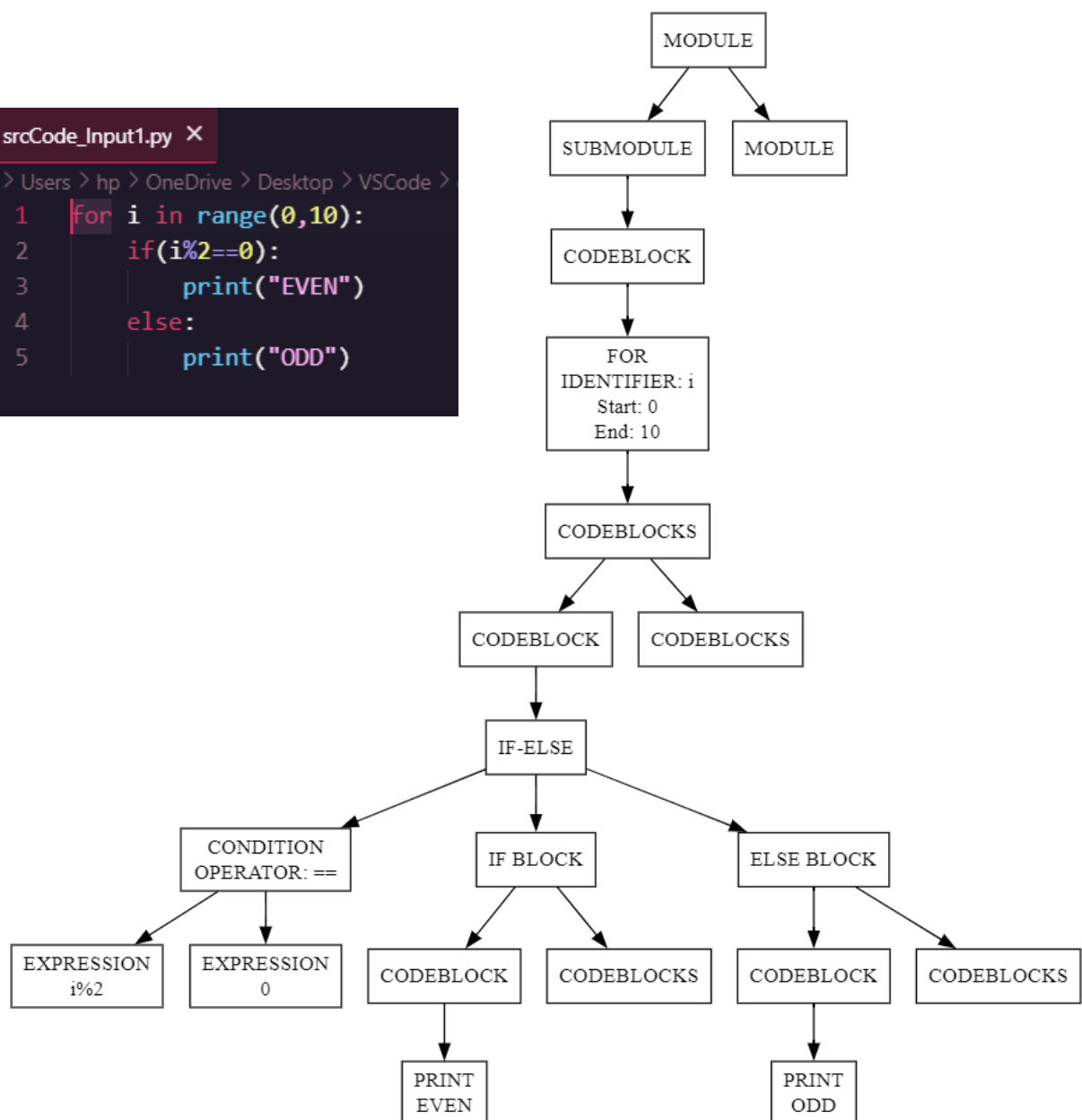
The parser takes the stream of tokens stored in LexerOutput.txt file as input and then matches it with the grammar of the language. The grammar is LL(1) in nature so the implemented parser is a predictive recursive descent parser. There exist corresponding functions to parse each of the non-terminals in the LL(1) grammar as constructed. Each node of the parse tree is implemented as a structure, which contains pointers to each of its direct descendants. If the parsing stage is successful, it outputs the

parse tree in the ParseTree.dot file, which can be visualized using dot file visualizers found online. In case it encounters any parsing error, it stops execution after returning the corresponding error message.

Test Cases and Outputs:

Test Case 1:

```
srcCode_Input1.py X
C: > Users > hp > OneDrive > Desktop > VSCode >
1  for i in range(0,10):
2      if(i%2==0):
3          print("EVEN")
4      else:
5          print("ODD")
```

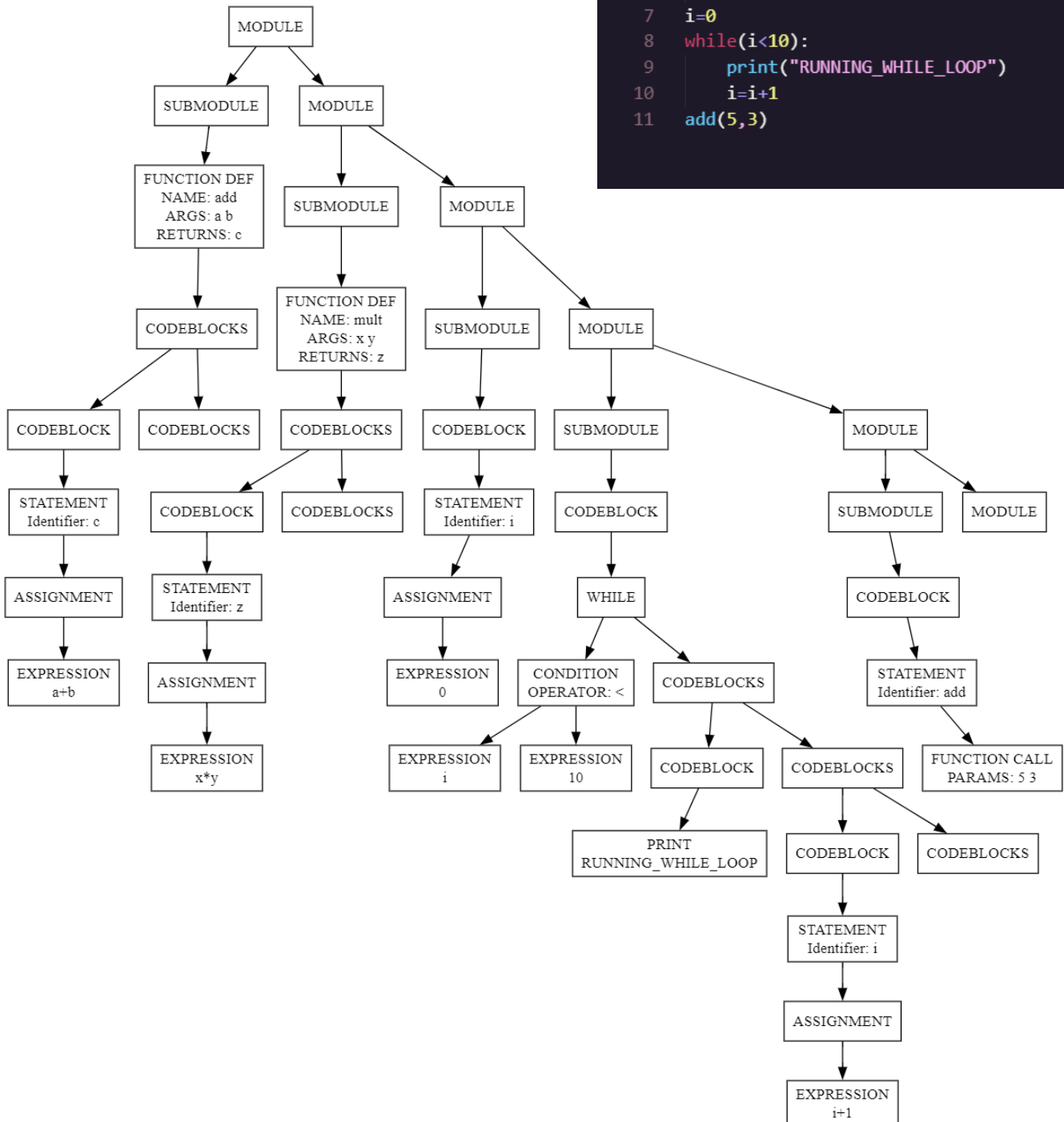


Test Case 2:

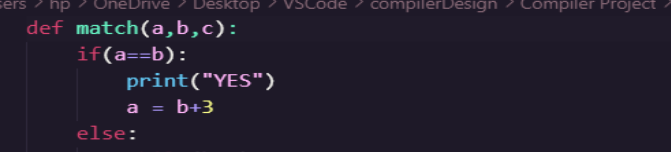
```

srcCode_Input1.py  srcCode_Input2.py X
C: > Users > hp > OneDrive > Desktop > VSCode > compilerDesign >
1  def add(a,b):
2      c=a+b
3      return c
4  def mult(x,y):
5      z=x*y
6      return z
7  i=0
8  while(i<10):
9      print("RUNNING_WHILE_LOOP")
10     i=i+1
11     add(5,3)

```



Test Case 3:



```
C:\>Users>hp>OneDrive>Desktop>VSCode>compilerDesign>Compiler Project>Compiler P
1 def match(a,b,c):
2     if(a==b):
3         print("YES")
4         a = b+3
5     else:
6         while(b<c):
7             b = b+1
8             match(a,b,c)
9         print("NO")
10 x = 1
11 for r in range(1,100):
12     x=x*r
13     if(x<1000000007):
14         x=x+1
15     else:
16         x=x%1000000007
17     match(x,r,2)
18 print("COMPLETE")
```

