Programming Language Concepts
Lab 4 Instructions

Y2024 – 01/04/2024, Matthieu DE MARI

Introduction

This PDF file contains the instructions for the Lab 4 of the 50.051 Programming Language Concepts course at the Singapore University of Technology and Design.

It consists of several tasks, which build up progressively. Each task has questions leading to the expected solution, which will sometimes have to be coded.

There are 9 questions to answer, for a total of 10 points.

You might work in groups of 2, maximum, or on your own. Larger groups sizes will not be allowed.

You will NOT be submitting this Lab 4 to eDimension.

Instead, you will need to show your answers to either the instructor or the TA in the room to get a checkoff, before the end of the session.

No other form of submission for this Lab 4 will be accepted.

Task 1 – A simple CFG code with basic production rules (1pt, 15min)

This first task aims to introduce students to a simple structure and representation of Context-Free Grammars (CFGs) in C. It can be seen as a simpler version of what is required for Lab 3.

You are provided with a starter code (in the CFG.c file) that includes basic functionalities required to create and manipulate some simple CFGs.

This first task lays the groundwork for understanding how CFGs can later be used in implementing top-down parsing algorithms.

For this task and the next two tasks, we will be relying on the following CFG (or a variation of it).

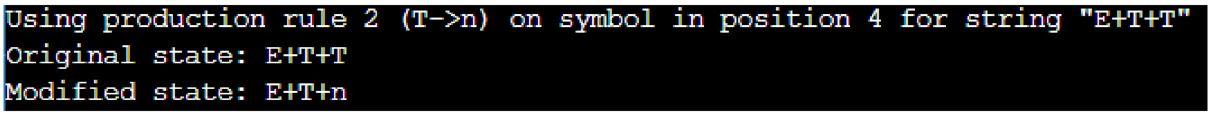
$$E \rightarrow E+T$$

$$E \rightarrow T$$

$$T \rightarrow number$$

Your job in this first task is to study the code in CFG.c file to get familiar with how production rules are created and used in this activity. There is no incomplete code for you to work on in the provided CFG.c file.

Running the code should produce the following output



```
Using production rule 2 (T->n) on symbol in position 4 for string "E+T+T"
Original state: E+T+T
Modified state: E+T+n
```

Question 1 (1pt): How would you modify the code if you wanted to apply production rule "E→E+T" on the non-terminal symbol "E" of the string "E+T+T"?

This concludes Task 1.

Task 2 – Parser with BFS (3pts, 30min)

In this task, we will implement a basic version of a non-predictive top-down parser using Breadth-First Search (BFS). The primary goal here is to illustrate the concept of BFS parsing and its limitations.

You may now open the BFS.c file and study the code inside of it.

It includes the previous structs and functions related to production rules. It also includes an implementation of a queue struct and functions to add elements to the queue (defined as the function enqueue) or retrieve the first element of the queue (defined as the function dequeue) updating the queue in the process, each time.

We also provide an incomplete prototype for the bfsParse function. As highlighted in the code, the function expects several parameters:

- startSymbol: the start symbol for our CFG (here, "E").

- targetString: the sequence of terminals we are trying to match (e.g. "n+n").

- rules: our array of production rules for the CFG.

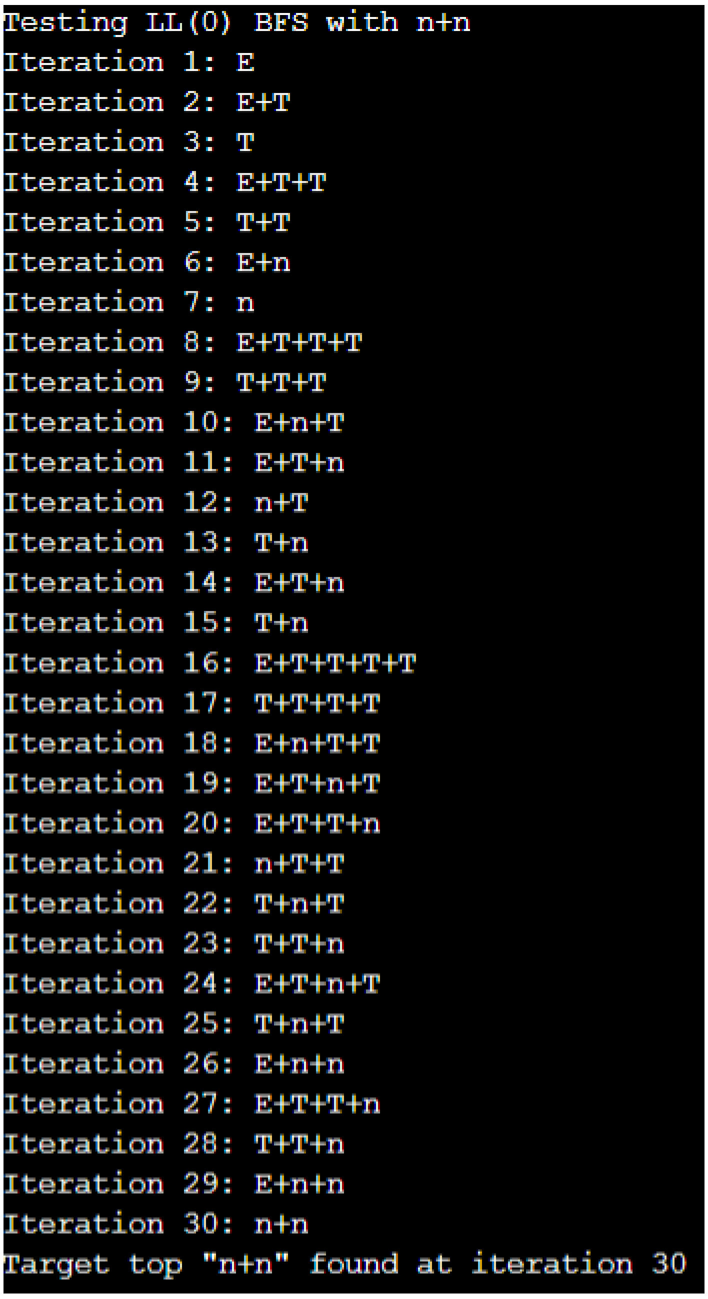
- numRules: the number of production rules.

It should implement a BFS search, as described in W10S1, slides 9-29. You may use the queue structure to facilitate the implementation of this function.

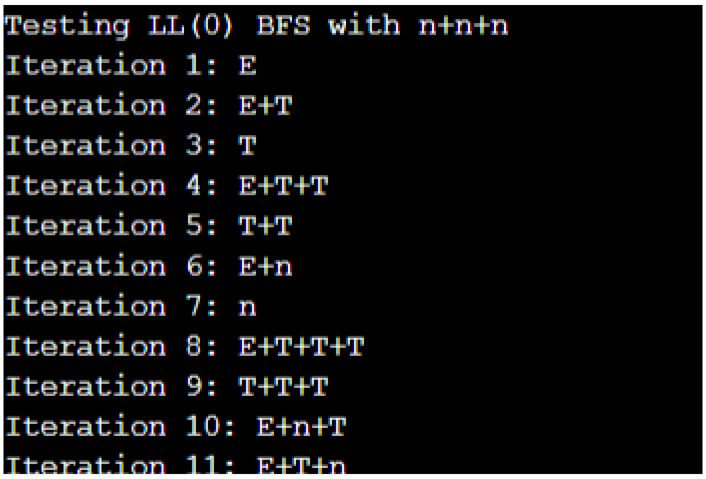
Question 2 (2pt): Show your final version of the bfsParse function for evaluation.

Question 3 (1pt): What is the main issue when using BFS? Can you show how this algorithm struggles to parse CFGs in certain configurations? What is the reason that explains this limitation?

After your implementation is complete, you should see the following result, when running the code.

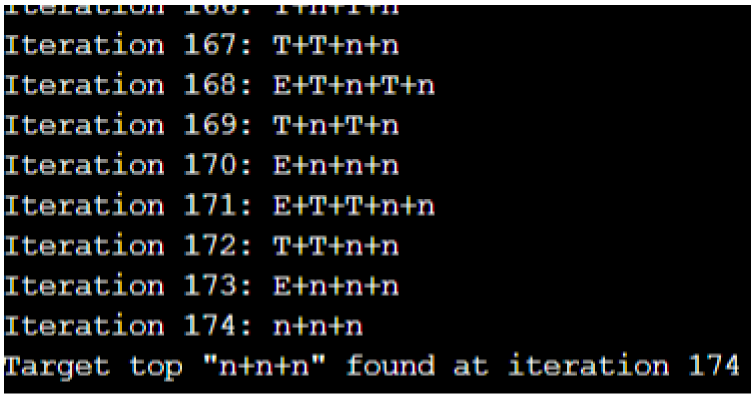


```
Testing LL(0) BFS with n+n
Iteration 1: E
Iteration 2: E+T
Iteration 3: T
Iteration 4: E+T+T
Iteration 5: T+T
Iteration 6: E+n
Iteration 7: n
Iteration 8: E+T+T+T
Iteration 9: T+T+T
Iteration 10: E+n+T
Iteration 11: E+T+n
Iteration 12: n+T
Iteration 13: T+n
Iteration 14: E+T+n
Iteration 15: T+n
Iteration 16: E+T+T+T+T
Iteration 17: T+T+T+T
Iteration 18: E+n+T+T
Iteration 19: E+T+n+T
Iteration 20: E+T+T+n
Iteration 21: n+T+T
Iteration 22: T+n+T
Iteration 23: T+T+n
Iteration 24: E+T+n+T
Iteration 25: T+n+T
Iteration 26: E+n+n
Iteration 27: E+T+T+n
Iteration 28: T+T+n
Iteration 29: E+n+n
Iteration 30: n+n
Target top "n+n" found at iteration 30
```

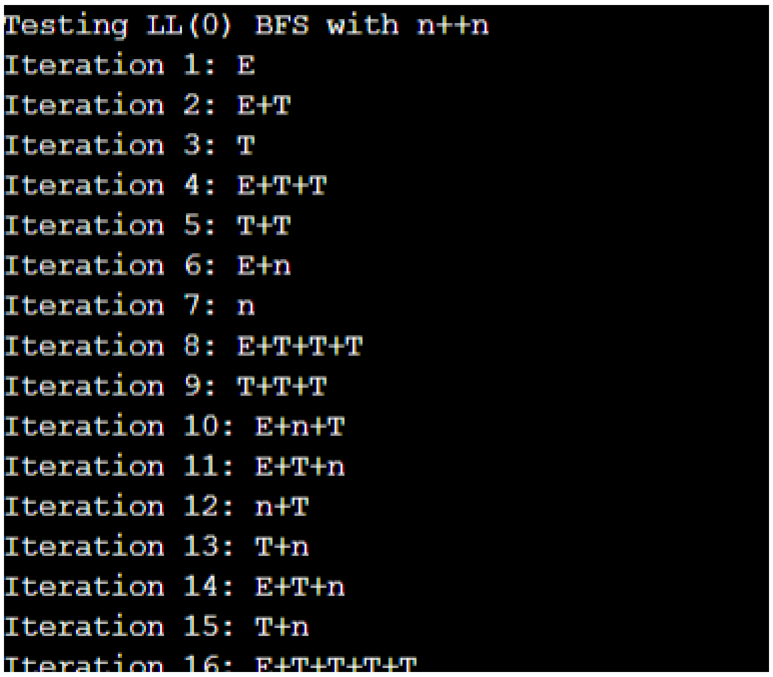


```
Testing LL(0) BFS with n+n+n
Iteration 1: E
Iteration 2: E+T
Iteration 3: T
Iteration 4: E+T+T
Iteration 5: T+T
Iteration 6: E+n
Iteration 7: n
Iteration 8: E+T+T+T
Iteration 9: T+T+T
Iteration 10: E+n+T
Iteration 11: E+T+n
```

…

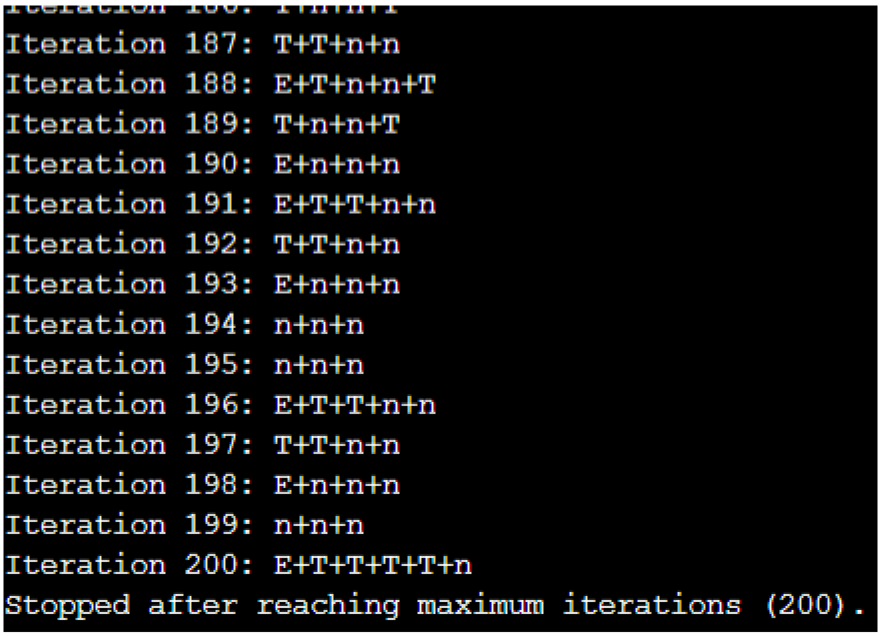


```
Iteration 166: T+n+T+n
Iteration 167: T+T+n+n
Iteration 168: E+T+n+T+n
Iteration 169: T+n+T+n
Iteration 170: E+n+n+n
Iteration 171: E+T+T+n+n
Iteration 172: T+T+n+n
Iteration 173: E+n+n+n
Iteration 174: n+n+n
Target top "n+n+n" found at iteration 174
```



```
Testing LL(0) BFS with n++n
Iteration 1: E
Iteration 2: E+T
Iteration 3: T
Iteration 4: E+T+T
Iteration 5: T+T
Iteration 6: E+n
Iteration 7: n
Iteration 8: E+T+T+T
Iteration 9: T+T+T
Iteration 10: E+n+T
Iteration 11: E+T+n
Iteration 12: n+T
Iteration 13: T+n
Iteration 14: E+T+n
Iteration 15: T+n
Iteration 16: E+T+T+T+T
```

...



```
Iteration 186: T+n+n+T
Iteration 187: T+T+n+n
Iteration 188: E+T+n+n+T
Iteration 189: T+n+n+T
Iteration 190: E+n+n+n
Iteration 191: E+T+T+n+n
Iteration 192: T+T+n+n
Iteration 193: E+n+n+n
Iteration 194: n+n+n
Iteration 195: n+n+n
Iteration 196: E+T+T+n+n
Iteration 197: T+T+n+n
Iteration 198: E+n+n+n
Iteration 199: n+n+n
Iteration 200: E+T+T+T+T+n
Stopped after reaching maximum iterations (200).
```

This concludes Task 2.

Task 3 – LL(0) Parser with DFS (3pts, 30min)

In this task, we will implement a basic version of a non-predictive top-down parser using Leftmost Depth-First Search (DFS). The primary goal here is to illustrate the concept of DFS parsing and its limitations.

You may now open the DFS.c file and study the code inside of it.

It includes the previous structs and functions related to production rules. It also includes an implementation of a stack struct and functions to add elements to the stack (defined as the function push) or retrieve the first element of the stack (defined as the function pop) updating the stack in the process, each time.

We also provide an incomplete prototype for the dfsParse function. As highlighted in the code, the function expects several parameters:

- startSymbol: the start symbol for our CFG (here, "E").

- targetString: the sequence of terminals we are trying to match (e.g. "n+n").

- rules: our array of production rules for the CFG.

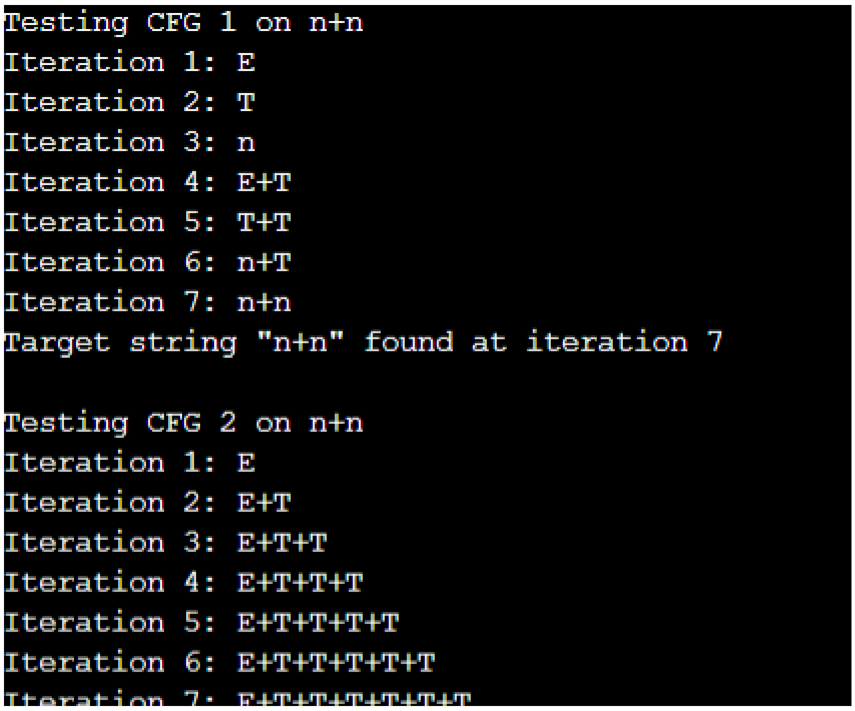
- numRules: the number of production rules.

It should implement a Leftmost DFS search, as described in W10S1, slides 47-58. You may use the stack structure to facilitate the implementation of this function.

Question 4 (2pt): Show your final version of the dfsParse function for evaluation.

Question 5 (1pt): What is the main issue when using Leftmost DFS? Can you show how this algorithm struggles to parse CFGs in certain configurations? What is the reason that explains this limitation?

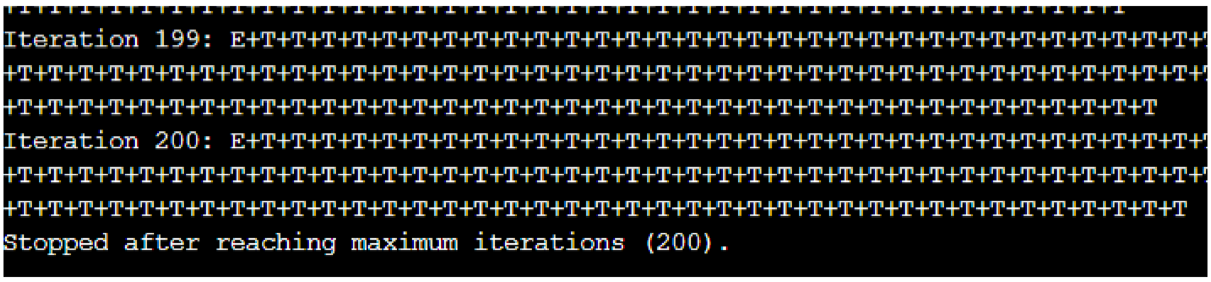
After your implementation is complete, you should see the following result, when running the code.



```
Testing CFG 1 on n+n
Iteration 1: E
Iteration 2: T
Iteration 3: n
Iteration 4: E+T
Iteration 5: T+T
Iteration 6: n+T
Iteration 7: n+n
Target string "n+n" found at iteration 7

Testing CFG 2 on n+n
Iteration 1: E
Iteration 2: E+T
Iteration 3: E+T+T
Iteration 4: E+T+T+T
Iteration 5: E+T+T+T+T
Iteration 6: E+T+T+T+T+T
Iteration 7: E+T+T+T+T+T+T
```

…



```
+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T
Iteration 199: E+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+
+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+
+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T
Iteration 200: E+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+
+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+
+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T+T
Stopped after reaching maximum iterations (200).
```

This concludes Task 3.

Task 4 – LL(1) Parser with LL(1) parsing table (3pts, 45min)

In this final part, we will try to implement our own function for an LL(1) parser, using an LL(1) parsing table of some sort. The primary goal here is to illustrate the concept of LL(1) parsing and how it might be able to figure out the correct derivation, without hesitation or backtracking. This LL(1) parsing algorithm relies on the ideas discussed in W10S1 materials, slides 72-100.

For this task 4, and this task 4 only, we will be using the CFG below.

$$E \rightarrow number$$

$$E \rightarrow (EOE)$$

$$O \rightarrow +$$

You may now open the LL1.c file and study the code inside of it.

You will recognize a few new functions, which have been provided to you.

First, the functions isTerminal and isNonTerminal, which will allow us to check if a character is a terminal or non-terminal symbol. These functions are complete and no additional modifications are needed.

Second, the function selectRule, which will receive the current leftmost non-terminal symbol to transform as currentChar and the next terminal symbol to match, targetChar. It should implement the logic of the LL(1) table, returning the production rule to use to transform currentChar into something new that can eventually match the next non-terminal targetChar. This function is incomplete and will require that you figure out the LL(1) table for the CFG above.

Question 6 (1pt): Show your LL(1) table for the CFG above.

Question 7 (1pt): Show your final version of the selectRule function for evaluation.

Eventually, you will have to resolve the missing code in the function LL1Parse. As before, it will receive the same four parameters:

- startSymbol: the start symbol for our CFG (here, "E").

- targetString: the sequence of terminals we are trying to match (e.g. "n+n").

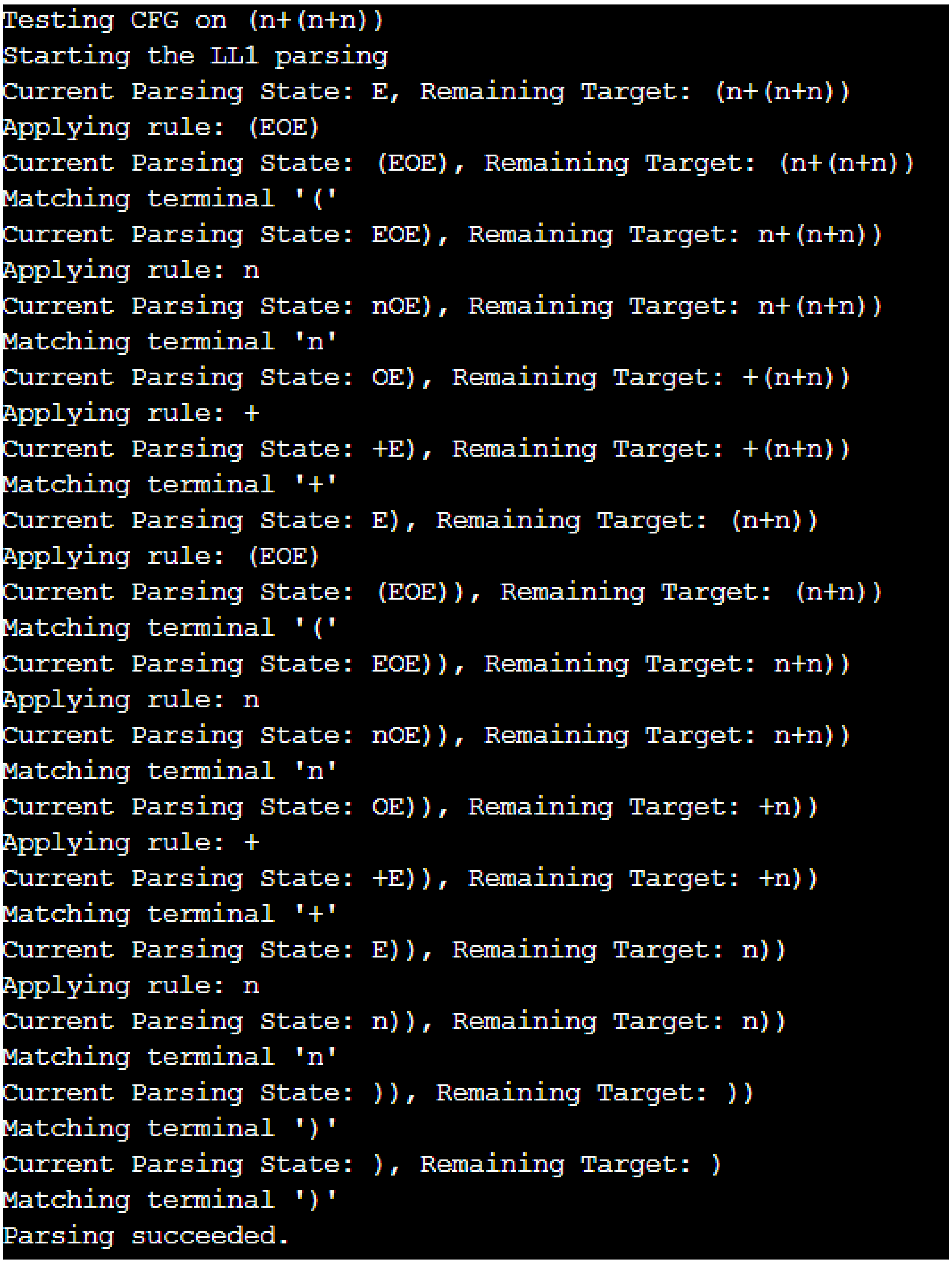
- rules: our array of production rules for the CFG.

- numRules: the number of production rules.

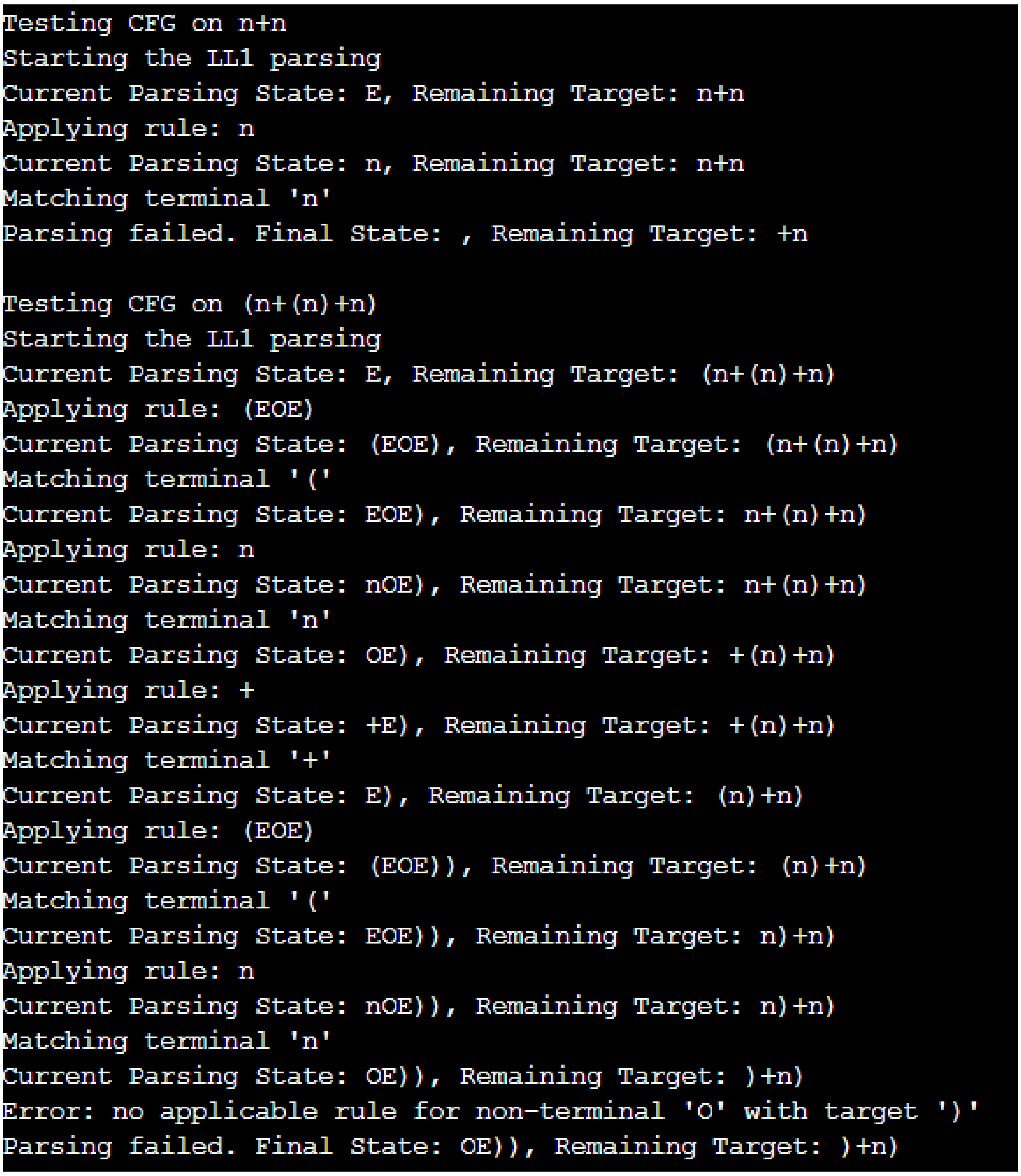
It should then implement an LL(1) parsing algorithm, which will correctly guess the right derivation to use, without backtracking.

Question 8(1pt): Show your final version of the LL1Parse function for evaluation.

After your implementation is complete, you should see the following result, when running the code.



```
Testing CFG on (n+(n+n))
Starting the LL1 parsing
Current Parsing State: E, Remaining Target: (n+(n+n))
Applying rule: (EOE)
Current Parsing State: (EOE), Remaining Target: (n+(n+n))
Matching terminal '('
Current Parsing State: EOE), Remaining Target: n+(n+n))
Applying rule: n
Current Parsing State: nOE), Remaining Target: n+(n+n))
Matching terminal 'n'
Current Parsing State: OE), Remaining Target: +(n+n))
Applying rule: +
Current Parsing State: +E), Remaining Target: +(n+n))
Matching terminal '+'
Current Parsing State: E), Remaining Target: (n+n))
Applying rule: (EOE)
Current Parsing State: (EOE)), Remaining Target: (n+n))
Matching terminal '('
Current Parsing State: EOE)), Remaining Target: n+n))
Applying rule: n
Current Parsing State: nOE)), Remaining Target: n+n))
Matching terminal 'n'
Current Parsing State: OE)), Remaining Target: +n))
Applying rule: +
Current Parsing State: +E)), Remaining Target: +n))
Matching terminal '+'
Current Parsing State: E)), Remaining Target: n))
Applying rule: n
Current Parsing State: n)), Remaining Target: n))
Matching terminal 'n'
Current Parsing State: )), Remaining Target: ))
Matching terminal ')'
Current Parsing State: ), Remaining Target: )
Matching terminal ')'
Parsing succeeded.
```



```
Testing CFG on n+n
Starting the LL1 parsing
Current Parsing State: E, Remaining Target: n+n
Applying rule: n
Current Parsing State: n, Remaining Target: n+n
Matching terminal 'n'
Parsing failed. Final State: , Remaining Target: +n

Testing CFG on (n+(n)+n)
Starting the LL1 parsing
Current Parsing State: E, Remaining Target: (n+(n)+n)
Applying rule: (EOE)
Current Parsing State: (EOE), Remaining Target: (n+(n)+n)
Matching terminal '('
Current Parsing State: EOE), Remaining Target: n+(n)+n)
Applying rule: n
Current Parsing State: nOE), Remaining Target: n+(n)+n)
Matching terminal 'n'
Current Parsing State: OE), Remaining Target: +(n)+n)
Applying rule: +
Current Parsing State: +E), Remaining Target: +(n)+n)
Matching terminal '+'
Current Parsing State: E), Remaining Target: (n)+n)
Applying rule: (EOE)
Current Parsing State: (EOE)), Remaining Target: (n)+n)
Matching terminal '('
Current Parsing State: EOE)), Remaining Target: n)+n)
Applying rule: n
Current Parsing State: nOE)), Remaining Target: n)+n)
Matching terminal 'n'
Current Parsing State: OE)), Remaining Target: )+n)
Error: no applicable rule for non-terminal 'O' with target ')'
Parsing failed. Final State: OE)), Remaining Target: )+n)
```

This concludes Task 4 and Lab 4.