50.051 Programming Language Concepts – Spring 2024

Atul Parida  
1006184

Submitted 6th April 2024

Answers to tasks are given below. Each task is started on a new page (though for Task 1 that’s a rather inefficient use of space, an antithesis to what C is all about):

Task 1:

1. The purpose of this CFG is to define the syntax of arithmetic expressions consisting of addition operations (+) between positive integers (represented by ‘n’ here). It lets us check whether a given string of symbols conforms to the CFG’s grammar rules representing a valid arithmetic expression. This is used to validate and analyse the input code, convert it into tokens, and determine if it’s a valid sequence or not within the compiler to make sure the input follows the compiler’s syntax rules.
2. If a non-terminal symbol can be replaced by a sequence including that symbol itself, the production rule is recursive. Eg: in rule 2, the symbol E appears in both the LHS and RHS of the expression. It allows us to describe structures that can infinitely repeat or nest, such as expression like “n+n+n” or “n+n+n+n…” and so on, where a valid expression is repeated. Without it, parsing nested structures would be a lot more cumbersome.

Task 2:

1. The following code is present within “Task2.h”:  
   A screen shot of a computer code

   Description automatically generated

|  |  |  |
| --- | --- | --- |
| Symbol | Is it terminal? | Is it a Start Symbol? |
| S | Non-terminal (0) | Start symbol (1) |
| E | Non-terminal (0) | Not a start symbol (0) |
| T | Non-terminal (0) | Not a start symbol (0) |
| + | Terminal (1) | Not applicable (0) |
| n | Terminal (1) | Not applicable (0) |

1. I’ve given the symbols and their characteristics in a table:
2. The following code is present within “Task2.c”:

A screen shot of a computer code

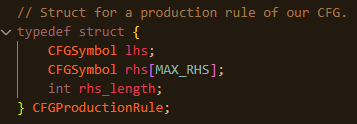
Description automatically generated

1. The following code is present within "Task2.c”:

A black background with colorful text

Description automatically generated

1. The following code is present within “Task2.h”:



1. For a production rule to be valid, all symbols in the LHS should be non-terminal in nature.
2. The following code is present within “Task2.c”:  
   A computer screen with colorful text

   Description automatically generated
3. The following code is present within “Task2.c”:
4. A computer screen shot of code

   Description automatically generated  
   It outputs the correct output when run, as outlined in the Lab 3 instructions.
5. The following code is present within “Task2.h”:  
   A screen shot of a computer code

   Description automatically generated
6. The following code is present within "Task2.c”:  
   A computer screen shot of code

   Description automatically generated
7. The following code is present within “Task2.c”:  
   A computer code on a black background

   Description automatically generated  
   This outputs the correct print as required in the instructions to Lab 3.

Task 3:

1. We import the init\_Terminal() function in the Tokenizer because it deals with terminal symbols, which are the focus of tokenization. The init\_NonTerminal() function is not needed because the Tokenizer does not handle non-terminals or the CFG structure.
2. ^[1-9][0-9]\*$ would suffice as a valid RegEx to our problem.
3. Since the character set is limited, we don’t need a RegEx for tokenization.  
     
   Here’s a simple pseudocode for the maximal munch algorithm specific to this problem:  
     
   *tokens 🡨 [ ]  
   string\_buffer 🡸 new StringBuffer(null)*

*for each character c in the input string:*

*if c is a digit:*

*string\_buffer.append(c)*

*else if string\_buffer != null:*

*converted\_buffer 🡨 tokenize(string\_buffer)  
 tokens.append(converted\_buffer)*

*string\_buffer 🡨 null*

*string\_buffer.append(c)*

*else:*

*raise an error ideally, or skip the character if we don’t have error handling*

*if string\_buffer != null:*

*converted\_buffer 🡨 tokenize(string\_buffer)  
tokens.append(converted\_buffer)*

*return tokens*

1. The following code is present within “Task3.c”:

A computer screen shot of a code

Description automatically generated

Task 4:

1. A valid derivation for “n+n+n” using our CFG from Task 1 is as follows:

Rule 1, position 0: *S 🡪 E*

Rule 2, position 1: *E 🡪 E + T*

Rule 2, position 1: *E + T 🡪 E + E + T*Rule 3, position 1: *E + E + T 🡪 E + T + T*

Rule 3, position 0: *E + E + T 🡪 T + T + T*

Rule 4, position 0: *T + T + T 🡪 n + T + T*

Rule 4, position 1: *n + T + T 🡪 n + n + T*

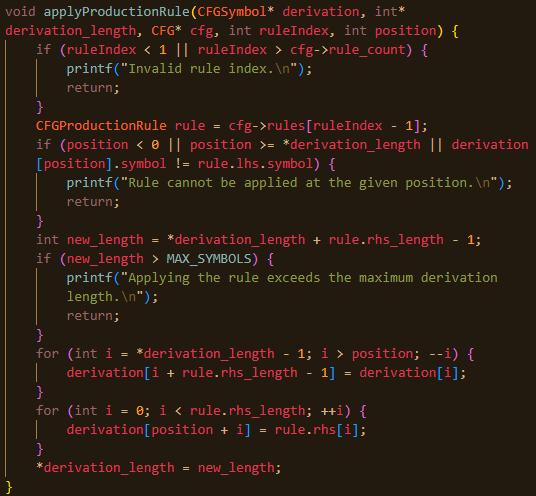
Rule 4, position 2: *n + n + T 🡪 n + n + n*

1. The following code is present within “Task4.c”:

A screen shot of a computer code

Description automatically generated

1. The following code is present within “Task4.c”:



1. The following code is present within “Task4.c”:

A screen shot of a computer code

Description automatically generated