# CS410 Project 2
# Design Report

İsmail Can Yağmur

December 2022

## Introduction

This reports discuss the implementation of CFG to CNF conversion written in C++17 with g++ compiler. Context Free Grammers written in a specific format in text files are read and processed in design logic.

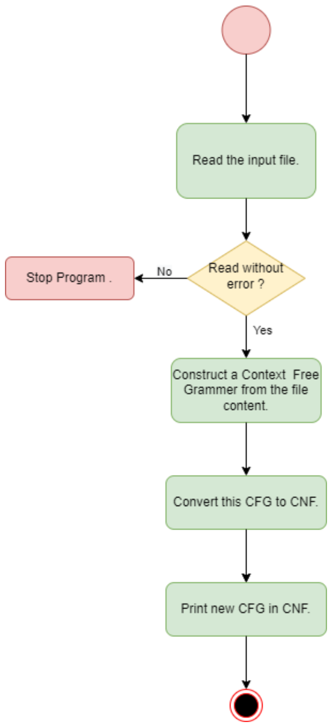
## Objectives

The implementation follows below objectives.

1. Read the input file.
2. Construct a Context Free Grammer from the file content.
3. Convert this CFG to CNF.
4. Print the new CFG in CNF to console in text file format.

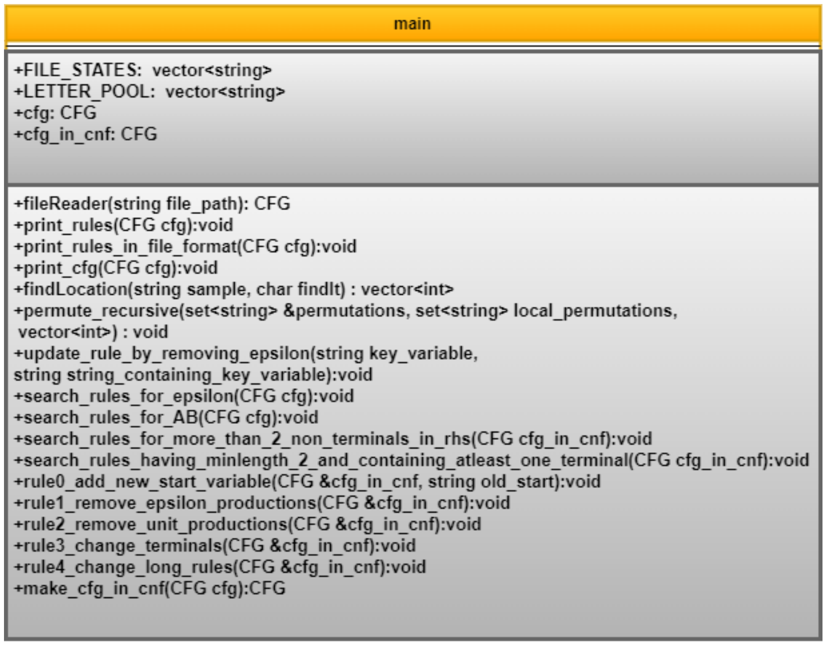
## Software Architecture Overview

### Activity Diagram



As it can be seen from the activity diagram, input file is read at the start of the program. If it is read with an error, the program prints the error to the console and exits. Otherwise, Context Free Grammer is constructed from the file content. After, created CFG is converted to its Chomsky normal form. At the end, new CFG in CNF is printed to the console in the same format that is used in input files. Also, each process of conversion is printed to the console to understand the details of conversion.

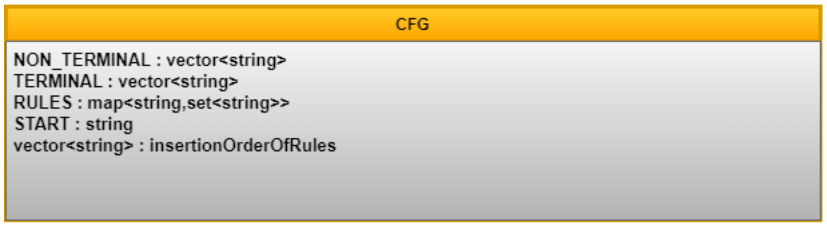
## Class Diagram



```
main

+FILE_STATES:  vector<string>
+LETTER_POOL:  vector<string>
+cfg: CFG
+cfg_in_cnf: CFG

+fileReader(string file_path): CFG
+print_rules(CFG cfg):void
+print_rules_in_file_format(CFG cfg):void
+print_cfg(CFG cfg):void
+findLocation(string sample, char findIt) : vector<int>
+permute_recursive(set<string> &permutations, set<string> local_permutations,
 vector<int>) : void
+update_rule_by_removing_epsilon(string key_variable,
string string_containing_key_variable):void
+search_rules_for_epsilon(CFG cfg):void
+search_rules_for_AB(CFG cfg):void
+search_rules_for_more_than_2_non_terminals_in_rhs(CFG cfg_in_cnf):void
+search_rules_having_minlength_2_and_containing_atleast_one_terminal(CFG cfg_in_cnf):void
+rule0_add_new_start_variable(CFG &cfg_in_cnf, string old_start):void
+rule1_remove_epsilon_productions(CFG &cfg_in_cnf):void
+rule2_remove_unit_productions(CFG &cfg_in_cnf):void
+rule3_change_terminals(CFG &cfg_in_cnf):void
+rule4_change_long_rules(CFG &cfg_in_cnf):void
+make_cfg_in_cnf(CFG cfg):CFG
```



```
CFG

NON_TERMINAL : vector<string>
TERMINAL : vector<string>
RULES : map<string,set<string>>
START : string
vector<string> : insertionOrderOfRules
```

This program consists of 1 class and 1 struct. Struct named CFG encapsulates the definition members of the Context Free Grammers. Also, it has a string variable named insertionOrderOfRules to print the rules in the order they put into the map. Context Free Grammer definition text files are read thanks to **FileReader** method. Each state in the text file representing which definition member of the grammar will be captured thanks to a vector of strings containing file states called as **FILE_STATES**. While reading the content of the file, **CFG** structure is used to encapsulate the definition members of grammer. This structure gives the blueprint of the desired grammer. These definition members of grammar are named in the code as follows : **NON_TERMINAL, TERMINAL, RULES and START**. Also, **NON_TERMINAL,TERMINAL** variables are encapsulated in vector data structure. **RULES** variable is encapsulated in map of string and set of string data structure. **START** variable is simply a string. After creating a CFG from the text file, **make_cfg_in_cnf** method is used for conversion. It converts the given CFG to its CNF format by dividing each step of the algorithm into different functions. After the conversion, the content of the CFG is displayed on the console in given file format thanks to **print_cfg** method.

## Conversion Process

While converting the CFG to CNF, functional programming is used. The conversion process is divided into 5 main parts. Each part is solved in the corresponding function. All these functions are executed in another function namely **make_cfg_in_cnf**. In the first step, a new start variable is added to CFG using **rule0_add_new_start_variable** function. In the second step, epsilon rules are removed and the other rules are updated using **rule1_remove_epsilon_productions** function. Basically, the function searches epsilon rules and updates the CFG with respect to these rules until there is no epsilon rule in the CFG. In the third step, all unit productions are removed using **rule2_remove_unit_productions** function. Internally, the function searches unit rules and updates the CFG with respect to these rules until there is no unit rule in the CFG. In the fourth step, all the rules containing terminal variables on the right-hand side are converted except the only terminal RHS case using **rule3_change_terminals** function. In the fifth and the last rule, all the long rules having more than 2 non-terminals in RHS are converted to 2 non-terminal strings using **rule4_change_long_rules**. After conversion, the resulting CFG is printed to the console.

## Results

Here, I tested my algorithm with given text files, G1.txt and G2.txt. The program takes the path of the text file as an argument and prints the original CFG,conversion process and converted CFG in CNF in the text file format to the console.

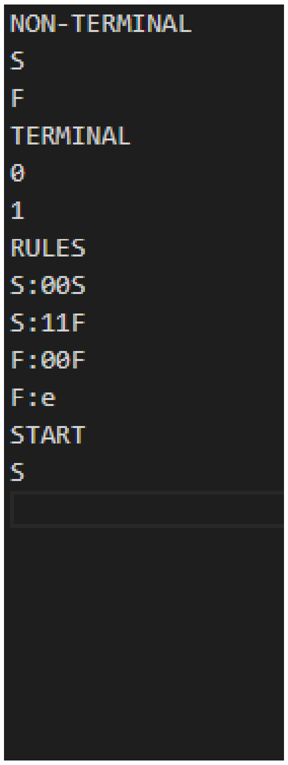
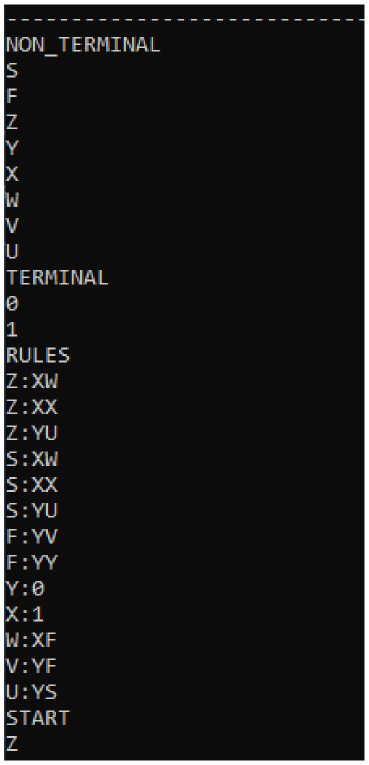
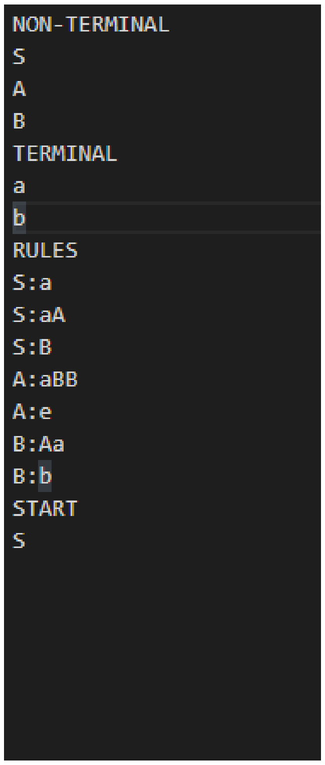
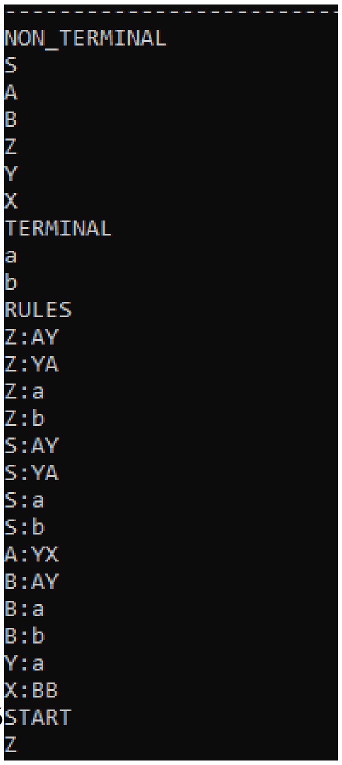
| CFG | CNF |
|---|---|
|  NON-TERMINAL<br>S<br>F<br>TERMINAL<br>0<br>1<br>RULES<br>S:00S<br>S:11F<br>F:00F<br>F:e<br>START<br>S |  ------------------------------<br>NON_TERMINAL<br>S<br>F<br>Z<br>Y<br>X<br>W<br>V<br>U<br>TERMINAL<br>0<br>1<br>RULES<br>Z:XW<br>Z:XX<br>Z:YU<br>S:XW<br>S:XX<br>S:YU<br>F:YV<br>F:YY<br>Y:0<br>X:1<br>W:XF<br>V:YF<br>U:YS<br>START<br>Z |
|  NON-TERMINAL<br>S<br>A<br>B<br>TERMINAL<br>a<br>b<br>RULES<br>S:a<br>S:aA<br>S:B<br>A:aBB<br>A:e<br>B:Aa<br>B:b<br>START<br>S |  ------------------------<br>NON_TERMINAL<br>S<br>A<br>B<br>Z<br>Y<br>X<br>TERMINAL<br>a<br>b<br>RULES<br>Z:AY<br>Z:YA<br>Z:a<br>Z:b<br>S:AY<br>S:YA<br>S:a<br>S:b<br>A:YX<br>B:AY<br>B:a<br>B:b<br>Y:a<br>X:BB<br>START<br>Z |

Table 1: Table of results