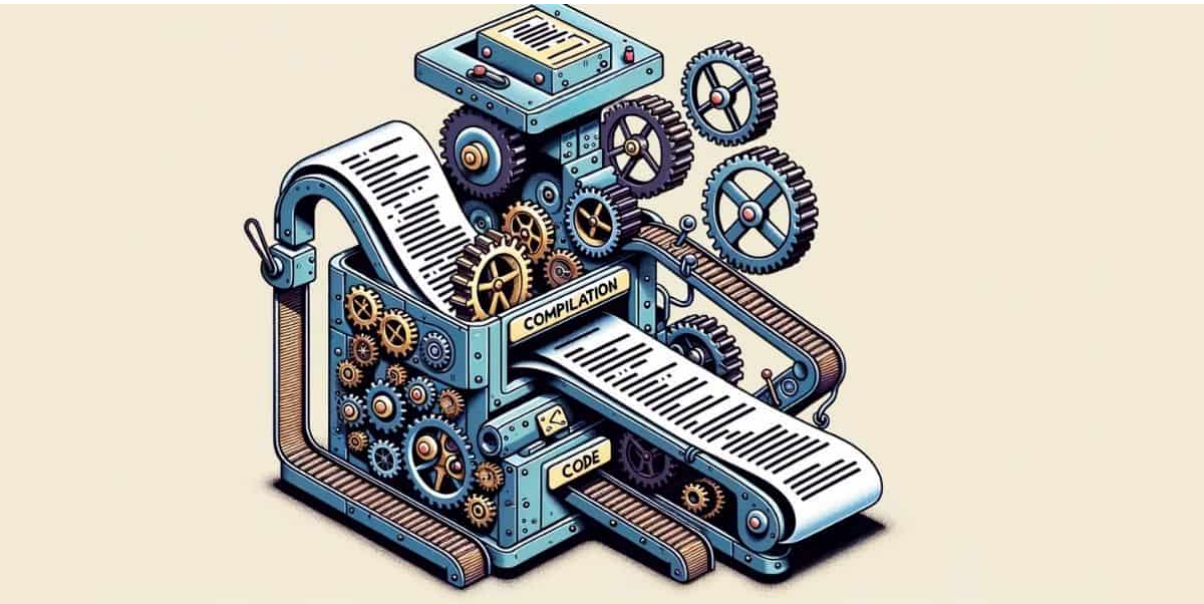
Transformation de grammaires algébriques en FNC et FNG + generation de mots : Rapport

BENSALEM NAZYM (22405066)
AHMED IACOB ESSALLAMI MANA (21926101)

Université de Versailles – L3 Informatique
2024-2025



Contents

Introduction

Ce rapport détaille tout le programme avec toutes les fonctions intermediaires, les exemples , le makefile, le lex, le main, les structures et la logique implementee.

1 Structure des données

Pour représenter une grammaire, nous avons utilisé deux structures principales :

- `Rule` : représente une règle avec un non-terminal et ses productions.
- `Grammaire` : représente l'ensemble des règles.

Listing 1: Déclaration des structures

```
#define MAX_RULES 100
#define MAX_SYMBOLS 100
#define MAX_NON_TERMINAUX 250

typedef struct {
    char non_terminal[MAX_SYMBOLS]; // Non-terminal
    char productions[MAX_RULES][MAX_SYMBOLS]; // Productions
    int production_count; // Nombre de productions
} Rule;

typedef struct {
    Rule rules[MAX_RULES]; // Ensemble des regles
    int rule_count; // Nombre de regles
} Grammaire;
```

2 Fonctions implémentées

2.1 Nettoyage des chaînes

La fonction `nettoyer_chaine` supprime les espaces dans une chaîne de caractères, facilitant ainsi le traitement des grammaires.

Listing 2: Nettoyage des chaînes

```
void nettoyer_chaine(char *str) {
    char *src = str, *dst = str;
    while (*src) {
        if (*src != ' ') {
            *dst++ = *src;
        }
        src++;
    }
    *dst = '\0';
}
```

2.2 Vérification des non-terminaux

La fonction `isNonTerminal` vérifie si un symbole est un non-terminal basé sur sa première lettre et la suivante et son format.

```
int isNonTerminal(const char *symbol) {
    return strlen(symbol) == 2 && isupper(symbol[0]) && isdigit(
       symbol[1]);
}
```

2.3 Vérification des terminaux

La fonction `isTerminal` vérifie si un symbole est un terminal (minuscule).

Listing 3: Vérification des terminaux

```
int isTerminal(char c) {
    return islower(c);
}
```

2.4 verifier si un non terminal existe dans notre grammaire(utile avant de generer)

La fonction $\texttt{non}_t erminal_e xistsvérifiesiunnon - terminalexistedéjàdanslagrammaire$

```
int non_terminal_exists(const Grammaire *grammaire,
const char *non_terminal) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        if (strcmp(grammaire->rules[i].non_terminal, non_terminal
           ) == 0) {
            return 1;
        }
    }
    return 0;
}
```

2.5 Génération des non-terminaux uniques

Pour gérer des grammaires complexes, nous avons besoin de générer des non-terminaux uniques. La fonction generate_non_terminal crée un nouveau non-terminal qui n'existe pas encore dans la grammaire. On commence par Z9 et on descend jusqu'à Z0, puis Y9, Y0... jusqu'à A9, A0. voici notre fonction:

```
void generate_non_terminal(char *result, const Grammaire *
   grammaire) {
    static int letter_index = 25; // Commencer par 'Z'
    static int number_index = 9;  // Commencer par 9
    int attempts = 0;

    printf("Début : letter_index=%d,
    number_index=%d\n", letter_index, number_index);

    do {
        snprintf(result, MAX_SYMBOLS, "%c%d", 'A' +
```

```
        letter_index, number_index);

        if (--number_index < 0) {
            number_index = 9;
            if (--letter_index < 0) {
                fprintf(stderr, "Erreur : Limite
                de non-terminaux atteinte (A0
                à Z9 épuisés).\n");
                exit(EXIT_FAILURE);
            }
        }

        attempts++;
        if (attempts > MAX_NON_TERMINAUX) {
            fprintf(stderr, "Erreur : Trop de
            tentatives pour générer un nouveau
            non-terminal.\n");
            exit(EXIT_FAILURE);
        }

        printf("Généré : %s\n", result); // Affiche le non-
           terminal généré
    } while (non_terminal_exists(grammaire, result));

    printf("Fin : letter_index=%d,
    number_index=%d\n",
    letter_index, number_index);
}
```

Notre fonction contient egalement des etapes de debeugage avec le chiffre et la lettre avant generation du non terminal , l'affichage du non terminal genere et la lettre et le chiffre apres la generation .

Malheureusement, nous n'avons pas pu implémenter la fonction souhaitée, qui génère un non-terminal et vérifie d'abord s'il existe déjà dans notre grammaire. Notre fonction semblait correcte, mais après plusieurs tests, elle ne fonctionnait pas. Nous vous présentons tout de même le code des fonctions qu'on avait realises pour réaliser cela :

Listing 4: Code pour générer des non-terminaux uniques

```
int non_terminal_exists(const Grammaire *grammaire,
const char *non_terminal) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        if (strcmp(grammaire->rules[i].non_terminal,
        non_terminal) == 0) {
            return 1;
        }
    }
    return 0;
}


void generate_non_terminal(char *result, const Grammaire *
```

```
    grammaire) {
     static char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
     static int current_letter_index = 0;
     static int current_digit = 0;

     while (1) {

         snprintf(result, MAX_SYMBOLS,
         "%c%d", letters
         [current_letter_index], current_digit);


         if (!non_terminal_exists(grammaire, result)) {
             break;
         }


         current_digit++;
         if (current_digit > 9) {
             current_digit = 0;
             current_letter_index++;
             if (letters[current_letter_index] == 'E') {
                 current_letter_index++;
             }
             if (current_letter_index >= 25) {
                 fprintf(stderr, "Erreur:Tous
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣les␣non␣terminaux
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣possibles␣sont␣utilisés.\n");
                 exit(EXIT_FAILURE);
             }
         }
     }
}
```

2.6 Lecture de la grammaire depuis un fichier

La fonction lire Grammaire lit une grammaire à partir d'un fichier texte et la stocke dans la structure Grammaire. Voici ses principales fonctionnalités :

- Ouvre et lit le fichier ligne par ligne.
- Nettoie chaque ligne en supprimant les espaces superflus.
- Parse chaque ligne pour extraire le non-terminal et ses productions.
- Définit automatiquement le premier non-terminal comme axiome si aucun n'est défini.
- Stocke chaque règle (non-terminal et ses productions) dans la structure Grammaire.

Listing 5: Lecture de la grammaire depuis un fichier

```
int lire_grammaire(Grammaire *grammaire, const char *filename) {
    grammaire->rule_count = 0;
```

```
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return -1;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        line[strcspn(line, "\n")] = '\0'; // Supprime le saut de
            ligne
        nettoyer_chaine(line);           // Nettoyer les espaces
            inutiles

        if (strlen(line) == 0) {
            continue;
        }

        Rule rule;
        rule.production_count = 0;

        char *token = strtok(line, ":");
        if (token == NULL) {
            fprintf(stderr, "Erreur : Format incorrect : %s\n",
                line);
            fclose(file);
            return -1;
        }
        strcpy(rule.non_terminal, token);

        token = strtok(NULL, "|");
        while (token != NULL) {
            nettoyer_chaine(token); // Nettoyer chaque production
            strcpy(rule.productions[rule.production_count++],
                token);
            token = strtok(NULL, "|");
        }

        grammaire->rules[grammaire->rule_count++] = rule;
    }

    fclose(file);
    return 0;
}
```

2.7 Affichage de la grammaire

La fonction afficher_grammaire est destinée à afficher une représentation textuelle de la grammaire. Elle parcourt les règles de la grammaire et les imprime dans un format lisible.qui est cote axiome ->prod1 | prod2 |.... 2eme membre droit -> prod1 | .... etc...

```
void afficher_grammaire(Grammaire *grammaire) {
    printf("Grammaire:\n");
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        printf("%s␣->␣", rule->non_terminal);
        for (int j = 0; j < rule->production_count; j++) {
            printf("%s", rule->productions[j]);
            if (j < rule->production_count - 1) printf("␣|␣");
        }
        printf("\n");
    }
}
```

2.8 nettoyage des regles similaires

La fonction regrouper_terminaux factorise des regles qui donnent le meme terminal par exemple si on a A0->a B0->a ca cree A1->a et ca remplace le soccurrences de A0 et B0 par ce nouveau non terminal cree A1

```
void regrouper_terminaux(Grammaire *grammaire) {
    char terminal_to_non_terminal[128][MAX_SYMBOLS] = {{0}};
    // Associer chaque terminal à un unique non-terminal
    char non_terminals_to_replace[MAX_RULES][MAX_SYMBOLS];
    // Liste des anciens non-terminaux à remplacer
    char terminal_for_non_terminal[MAX_RULES][2] = {{0}};
    // Terminal associé à chaque ancien non-terminal
    int replace_count = 0;
    // Compteur des non-terminaux à remplacer
    Grammaire updated_grammaire = *grammaire;
    // Copie pour modification

    // Étape 1 : Identifier les terminaux similaires
    et créer un unique non-terminal pour chaque terminal
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        if (rule->production_count ==
        1 && strlen(rule->productions[0]) == 1 &&
        islower(rule->productions[0][0])) {
            char terminal = rule->productions[0][0];

            // Vérifier si un non-terminal existe déjà
            pour ce terminal
            if (strlen(terminal_to_non_terminal[
            (int)terminal]) == 0) {
                // Générer un nouveau non-terminal
                char new_non_terminal[MAX_SYMBOLS];
                generate_non_terminal(new_non_terminal, &
                   updated_grammaire);

                // Associer ce non-terminal au terminal
                strcpy(terminal_to_non_terminal
                [(int)terminal], new_non_terminal);
```

```
            // Ajouter une règle pour ce terminal
            Rule new_rule;
            strcpy(new_rule.non_terminal, new_non_terminal);
            snprintf(new_rule.productions[0],
            MAX_SYMBOLS, "%c", terminal);
            new_rule.production_count = 1;
            updated_grammaire.rules
            [updated_grammaire.rule_count++] = new_rule;
        }

        // Enregistrer l'ancien non-terminal à remplacer
        strcpy(non_terminals_to_replace[
        replace_count], rule->non_terminal);
        terminal_for_non_terminal[replace_count][0]
        = terminal;
        replace_count++;
    }
}

// Étape 2 : Supprimer les anciennes règles redondantes
Grammaire temp_grammaire = {0};
for (int i = 0; i < updated_grammaire.rule_count; i++) {
    Rule *rule = &updated_grammaire.rules[i];

    // Vérifier si c'est une règle redondante à supprimer
    int skip = 0;
    for (int j = 0; j < replace_count; j++) {
        if (strcmp
        (rule->non_terminal,
        non_terminals_to_replace[j]) == 0) {
            skip = 1;
            break;
        }
    }

    if (!skip) {
        temp_grammaire.rules[temp_grammaire.rule_count++]
        = *rule;
    }
}

updated_grammaire = temp_grammaire;

// Étape 3 : Mettre à jour toutes les règles avec
les nouveaux non-terminaux
for (int i = 0; i < updated_grammaire.rule_count;
i++) {
    Rule *rule = &updated_grammaire.rules[i];
    for (int j = 0; j < rule->production_count;
    j++) {
```

```
            char *prod = rule->productions[j];
            for (int k = 0; k < replace_count; k++) {
                if (strcmp(prod,
                non_terminals_to_replace[k]) == 0) {
                    strcpy(prod, terminal_to_non_terminal
                    [(int)terminal_for_non_terminal[k][0]]);
                }
            }
        }
    }

    // Étape 4 : Remplacer les anciens non-terminaux
    dans toutes les productions où ils apparaissent
    for (int i = 0; i < updated_grammaire.rule_count;
    i++) {
        Rule *rule = &updated_grammaire.rules[i];
        for (int j = 0; j < rule->production_count;
        j++) {
            char *prod = rule->productions[j];
            for (int k = 0; k < replace_count; k++) {
                // Si le non-terminal à remplacer
                est dans une production
                char *found = strstr(prod,
                   non_terminals_to_replace[k]);
                if (found) {
                    // Remplacer par le nouveau non-terminal
                    strncpy(found,
                    terminal_to_non_terminal
                    [(int)
                    terminal_for_non_terminal[k][0]], strlen
                    (non_terminals_to_replace[k]));
                }
            }
        }
    }

    // Mise à jour finale de la grammaire
    *grammaire = updated_grammaire;
}
```

2.9 sauvegarder une grammaire dans un fichier de sortie

La fonction sauvegarder_grammaire ecrit la grammaire passee en parametre dans un fichier de sortie , si le char c = 'g' ca l'ecrit dans le fichier mots.greibach sinon si cest 'c' alors ca l'ecrit dans mots.chomsky

```
void sauvegarder_grammaire(const Grammaire *grammaire,
const char *nom_base, char c) {
    // Construire le nom du fichier en fonction du caractère c
    char nom_fichier[MAX_SYMBOLS];
    if (c == 'c') {
```

```
        snprintf(nom_fichier, sizeof(nom_fichier),
        "%s.chomsky", nom_base);
    } else if (c == 'g') {
        snprintf(nom_fichier, sizeof(nom_fichier),
        "%s.greibach", nom_base);
    } else {
        printf("Erreur : caractère non valide. Utilisez 'c' ou 'g
           '.\n");
        return;
    }

    // Ouvrir le fichier en mode écriture
    FILE *fichier = fopen(nom_fichier, "w");
    if (!fichier) {
        perror("Erreur lors de l'ouverture du fichier");
        return;
    }

    // Parcourir les règles de la grammaire
    for (int i = 0; i < grammaire->rule_count; i++) {
        const Rule *rule = &grammaire->rules[i];

        // Écrire le non-terminal
        fprintf(fichier, "%s : ", rule->non_terminal);

        // Écrire les productions séparées par " | "
        for (int j = 0; j < rule->production_count; j++) {
            fprintf(fichier, "%s", rule->productions[j]);
            if (j < rule->production_count - 1) {
                fprintf(fichier, " | ");
            }
        }

        // Fin de ligne pour la règle
        fprintf(fichier, "\n");
    }

    // Fermer le fichier
    fclose(fichier);
    printf("Grammaire sauvegardée dans le fichier '%s'.\n",
       nom_fichier);
}
```

2.10 Reecriture des regles

dans notre logique nos productions sont toutes sur la meme ligne separees par des '|', donc si on a des productions qui sont sur plusieurs lignes differentes on doit les mettre sur la meme ligne

2.10.1 la fonction ajouter_production

La fonction ajouter_production permet d'ajouter une nouvelle production à une règle de la grammaire, mais elle vérifie d'abord que cette production n'est pas déjà présente. Si elle l'est, elle ne l'ajoute pas.

```
void ajouter_production(Rule *rule, const char *production) {
    for (int i = 0; i < rule->production_count; i++) {
        if (strcmp(rule->productions[i], production) == 0) {
            return; // Production déjà présente, on ne l'ajoute
               pas
        }
    }
    strcpy(rule->productions[rule->production_count++],
       production);
}
```

2.10.2 fonction pour réécrire la grammaire sous la forme avec "|" entre les productions pour chaque non-terminal

La fonction rewriter_grammaire réécrit la grammaire sous la forme où chaque non-terminal a ses productions séparées par des |, c'est-à-dire qu'elle fusionne les productions identiques dans une seule règle pour chaque non-terminal. Cela permet de simplifier la présentation de la grammaire, notamment pour les grammaires avec des productions alternatives.

```
void rewriter_grammaire(Grammaire *grammaire) {
    // Création d'un tableau pour stocker les nouvelles règles
    Grammaire nouvelle_grammaire;
    nouvelle_grammaire.rule_count = 0;

    // Parcours de chaque règle
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        // Chercher si une règle avec le même non-terminal
        existe déjà dans la nouvelle grammaire
        int found = 0;
        for (int j = 0; j < nouvelle_grammaire.rule_count;
        j++) {
            if (strcmp(nouvelle_grammaire.rules[j].
            non_terminal, rule->non_terminal) == 0) {
                // Ajouter toutes les productions de
                cette règle à la règle correspondante
                for (int k = 0; k < rule->production_count;
                k++) {
                    ajouter_production(&nouvelle_grammaire.
                    rules[j], rule->productions[k]);
                }
                found = 1;
                break;
            }
        }
```

```
        // Si la règle n'a pas encore été ajoutée,
        on l'ajoute␣à␣la␣nouvelle␣grammaire
␣␣␣␣␣␣␣␣if␣(!found)␣{
␣␣␣␣␣␣␣␣␣␣␣␣strcpy(nouvelle_grammaire.rules
␣␣␣␣␣␣␣␣␣␣␣␣[nouvelle_grammaire.rule_count].
␣␣␣␣␣␣␣␣␣␣␣␣non_terminal,␣rule->non_terminal);
␣␣␣␣␣␣␣␣␣␣␣␣nouvelle_grammaire.rules[nouvelle_grammaire.
␣␣␣␣␣␣␣␣␣␣␣␣rule_count].production_count␣=␣0;

␣␣␣␣␣␣␣␣␣␣␣␣for␣(int␣k␣=␣0;␣k␣<␣rule->production_count;
␣␣␣␣␣␣␣␣␣␣␣␣k++)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ajouter_production(&nouvelle_grammaire.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rules[nouvelle_grammaire.rule_count],
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rule->productions[k]);
␣␣␣␣␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣␣␣␣␣nouvelle_grammaire.rule_count++;
␣␣␣␣␣␣␣␣}
␣␣␣␣}

␣␣␣␣//␣Copier␣la␣nouvelle␣grammaire␣dans␣la␣grammaire␣d'origine
    *grammaire = nouvelle_grammaire;
}
```

2.11 suppression de E si il est colle a un terminal ou un non terminal dans une regle

pour cela on a cree 2 fonctions pour parcourir toutes les règles et, si un E est trouvé dans une production, vérifier s'il est suivi ou précédé par une majuscule ou une minuscule. Si c'est le cas, le E devrait être supprimé, et le reste de la production doit être conservé. on l'utilise dans chomsky et greibach apres la suppression de la recursivite gauche

```
bool est_majuscule_ou_minuscule(char c) {
    return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
}

void supprimer_E_non_isole(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        // Parcours de toutes les productions de chaque règle
        for (int j = 0; j < rule->production_count; j++) {
            char *production = rule->productions[j];
            int len = strlen(production);

            for (int i = 0; i < len; i++) {
                // On cherche les 'E' dans la production
                if (production[i] == 'E') {
                    // Vérifier si E est précédé ou suivi d'une
                       majuscule ou minuscule
```

```
                    if ((i > 0 && est_majuscule_ou_minuscule(
                        production[i - 1])) ||
                        (i < len - 1 &&
                            est_majuscule_ou_minuscule(production[
                            i + 1]))) {

                        // Supprimer le 'E' de la production
                        printf("Avant␣suppression␣de␣E␣:␣%s\n",
                            production);
                        // Décaler le reste de la chaîne après le
                            'E'
                        memmove(production + i, production + i +
                            1, len - i);
                        printf("Après␣suppression␣de␣E␣:␣%s\n",
                            production);
                        break; // Passer à la production suivante
                            après modification
                    }
                }
            }
        }
    }
}
```

3 fonctions de chomsky et greibach

3.1 factorisation de la grammaire

3.1.1 prefix_common_length

La fonction prefix_common_length Détermine la longueur du préfixe commun entre deux chaînes de caractères ex: abcA0 et abcA9 => |abc| = 3.

```
int prefix_common_length(const char *str1, const char *str2) {
    int len = 0;
    while (str1[len] != '\0' && str2[len] != '\0' && str1[len] ==
        str2[len])
    {
        len++;
    }
    return len;
}
```

3.1.2 factoriser_productions

La fonction factoriser_productions Factorise deux productions ayant un préfixe commun en introduisant un nouveau non-terminal.

```
int factoriser_productions(char *prod1, char *prod2, Grammaire *
   grammaire) {
    int prefix_len = prefix_common_length(prod1, prod2);

    if (prefix_len > 0) {
```

```
            char new_non_terminal[MAX_SYMBOLS];
            generate_non_terminal(new_non_terminal, grammaire);
            Rule new_rule;
            strcpy(new_rule.non_terminal, new_non_terminal);
            new_rule.production_count = 0;


            if (strlen(prod1) > prefix_len) {
                strcpy(new_rule.productions[new_rule.production_count
                   ++],
                prod1 + prefix_len);
            } else {
                strcpy(new_rule.productions[new_rule.
                production_count++], "E");
            }

            if (strlen(prod2) > prefix_len) {
                strcpy(new_rule.productions[new_rule.
                production_count++], prod2 + prefix_len);
            } else {
                strcpy(new_rule.productions[new_rule.
                production_count++], "E");
            }


            grammaire->rules[grammaire->rule_count++] = new_rule;


            snprintf(prod1, MAX_SYMBOLS, "%.*s%s",
            prefix_len, prod1, new_non_terminal);


            strcpy(prod2, prod1);

            return 1;
        }

        return 0;
    }
```

3.1.3 factoriser_rule

La fonction factoriser_rule Applique la factorisation à toutes les productions d'une règle donnée.

```
    void factoriser_rule(Rule *rule, Grammaire *grammaire) {
        for (int i = 0; i < rule->production_count; i++) {
            for (int j = i + 1; j < rule->production_count; j++) {

                if (factoriser_productions(rule->productions[i],
                rule->productions[j], grammaire)) {
```

```
                for (int k = j; k < rule->production_count - 1; k
                    ++) {
                    strcpy(rule->productions[k], rule->
                        productions[k + 1]);
                }
                rule->production_count--;
                j--; // Réexaminer la position actuelle
            }
        }
    }
}
```

3.1.4 factoriser

La fonction factoriser applique la factorisation à toutes les règles de la grammaire

```
void factoriser(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        factoriser_rule(rule, grammaire);
    }
}
```

3.2 suppression des regles de la forme X->E

La fonction supprimer_epsilon supprime les regles de la forme X->E sauf si X est l'axiome voici les etapes:

Étape 1 : Identifier les non-terminaux pouvant produire epsilon But : Identifier les nonterminaux qui peuvent générer epsilon (directement ou indirectement). Approche : On utilise un tableau epsilon_non_terminals pour marquer les nonterminaux capables de produire epsilon. On parcourt toutes les règles de la grammaire : Si une production est exactement "E", on marque le nonterminal comme pouvant produire epsilon. Si une production est composée uniquement de non-terminaux qui peuvent produire epsilon, on marque aussi le nonterminal correspondant. Ce processus est répété jusqu'à ce qu'il n'y ait plus de changements dans le tableau.

Étape 2 : Ajouter des variantes sans epsilon But : Pour chaque production, générer toutes les combinaisons possibles en remplaçant les occurrences des nonterminaux qui produisent epsilon par leur absence. Approche : Parcourir les productions de chaque règle. Pour chaque production, vérifier si elle contient un non-terminal pouvant produire epsilon. Si c'est le cas, générer une nouvelle production en retirant ce non-terminal (tout en gardant les autres parties de la production intactes). Ajouter la nouvelle production uniquement si elle n'existe pas déjà.

Étape 3 : Supprimer explicitement les productions epsilon But : Supprimer les productions contenant uniquement "E", sauf celles de l'axiome. Approche : Parcourir les règles. Supprimer toutes les productions "E" pour les non-terminaux qui ne sont pas l'axiome.

Étape 4 : Supprimer les règles inutiles But : Éliminer les règles qui n'ont plus de productions associées, car elles ne contribuent plus à la grammaire. Approche : Parcourir les règles. Si une règle ne contient aucune production et qu'elle n'est pas l'axiome, la supprimer. Vérifier également que les références à ce non-terminal dans d'autres productions sont supprimées.

Étape supplémentaire : Ajouter epsilon à l'axiome si nécessaire But : Si l'axiome

peut produire epsilon, s'assurer que cette production est explicitement présente.
Approche : Si l'axiome est marqué comme pouvant produire epsilon (dans epsilon$_n on_t erminals$)

```
void supprimer_epsilon(Grammaire *grammaire, const char *axiome)
   {
    int epsilon_non_terminals[MAX_RULES] = {0};
    int changes;

    // Étape 1
    do {
        changes = 0;
        for (int i = 0; i < grammaire->rule_count; i++) {
            if (epsilon_non_terminals[i]) continue;
            // Déjà marqué comme epsilon
            Rule *rule = &grammaire->rules[i];
            for (int j = 0; j < rule->production_count; j++) {
                if (strcmp(rule->productions[j], "E") == 0) {
                    epsilon_non_terminals[i] = 1;
                    changes = 1;
                    break;
                }
                // Vérifier si toutes les parties de
                la production peuvent produire epsilon
                int all_epsilon = 1;
                for (int k = 0; k < strlen(rule->
                productions[j]); k += 2) {
                    char non_terminal[3] =
                    {rule->productions[j][k],
                    rule->productions[j][k + 1], '\0'};
                    int found = 0;
                    for (int l = 0; l <
                    grammaire->rule_count; l++) {
                        if (strcmp(grammaire->rules[l].
                           non_terminal,
                        non_terminal) == 0) {
                            if (epsilon_non_terminals[l]) {
                                found = 1;
                                break;
                            }
                        }
                    }
                    if (!found) {
                        all_epsilon = 0;
                        break;
                    }
                }
                if (all_epsilon) {
                    epsilon_non_terminals[i] = 1;
                    changes = 1;
                    break;
                }
```

```
        }
    }
} while (changes);

// Étape 2
do {
    changes = 0; // Réinitialiser l'indicateur de
       modifications
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        int original_count = rule->production_count;

        for (int j = 0; j < original_count; j++) {
            char *prod = rule->productions[j];

            // Générer toutes les combinaisons en
            remplaçant les non-terminaux epsilon
            for (int k = 0; k <
            grammaire->rule_count; k++) {
                if (epsilon_non_terminals[k]) {
                    char *non_terminal = grammaire->rules[k].
                       non_terminal;
                    char *found = strstr(prod,
                    non_terminal);
                    while (found) {
                        char new_production
                        [MAX_SYMBOLS] = "";

                        // Partie avant le non-terminal
                        strncpy(new_production,
                        prod, found - prod);
                        new_production[found - prod]
                        = '\0';

                        // Partie après le non-terminal
                        strcat(new_production, found + strlen
                           (non_terminal));

                        // Ajouter la nouvelle
                        production si elle existe pas déjà
                        int exists = 0;
                        for (int l = 0; l < rule->
                           production_count; l++) {
                            if (strcmp(rule->productions[l],
                               new_production) == 0) {
                                exists = 1;
                                break;
                            }
                        }
                        if (!exists && strlen(new_production)
                            > 0) {
```

```
                                strcpy(rule->productions[rule->
                                    production_count++],
                                    new_production);
                                changes = 1;
                                // Une modification a été
                                effectuée
                            }

                            // Chercher la prochaine
                            occurrence
                            found = strstr(found + 1,
                                non_terminal);
                        }
                    }
                }
            }
        }
    } while (changes);

    // Étape 3
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        if (strcmp(rule->non_terminal, axiome) != 0) {
            for (int j = 0; j < rule->production_count;) {
                if (strcmp(rule->productions[j], "E") == 0) {
                    for (int k = j; k < rule->production_count -
                        1; k++) {
                        strcpy(rule->productions[k],
                        rule->productions[k + 1]);
                    }
                    rule->production_count--;
                } else {
                    j++;
                }
            }
        }
    }

    // Étape 4
    for (int i = 0; i < grammaire->rule_count;) {
        Rule *rule = &grammaire->rules[i];
        if (rule->production_count == 0
        && strcmp(rule->non_terminal,
        axiome) != 0) {
            char non_terminal_to_remove[MAX_SYMBOLS];
            strcpy(non_terminal_to_remove,
            rule->non_terminal);

            // Supprimer cette règle
            for (int j = i; j < grammaire->rule_count - 1;
            j++) {
```

```
            grammaire->rules[j] =
            grammaire->rules[j + 1];
        }
        grammaire->rule_count--;

        // Supprimer les références
        dans les autres règles
        for (int j = 0; j < grammaire->rule_count; j++) {
            Rule *other_rule = &grammaire->rules[j];
            for (int k = 0; k <
            other_rule->production_count;) {
                if (strstr(other_rule->
                productions[k],
                non_terminal_to_remove)) {
                    for (int l = k; l <
                    other_rule->production_count - 1;
                    l++) {
                        strcpy(other_rule->
                        productions[l],
                        other_rule->
                        productions[l + 1]);
                    }
                    other_rule->production_count--;
                } else {
                    k++;
                }
            }
        }
    } else {
        i++;
    }
}

// Étape supplémentaire
for (int i = 0; i < grammaire->rule_count; i++) {
    Rule *rule = &grammaire->rules[i];
    if (strcmp(rule->non_terminal, axiome) == 0) {
        if (epsilon_non_terminals[i]) {
            int already_has_epsilon = 0;
            for (int j = 0; j < rule->production_count; j++)
               {
                if (strcmp(rule->productions[j], "E") == 0) {
                    already_has_epsilon = 1;
                    break;
                }
            }
            if (!already_has_epsilon) {
                strcpy(rule->productions[rule->
                   production_count++], "E");
            }
        }
```

```
        }
    }
}
```

c'est sans hesitations la fonction qui nous a pris le plus de temps (la gestion de tous les cas quand on a plusieurs non terminaux qui s'annulent .....)

3.3 suppression des regles unite de la forme X->Y

La fonction supprimer_unite sert à éliminer les règles unitaires d'une grammaire. Une règle unitaire est une production qui ne contient qu'un seul non-terminal, par exemple : X0 -> Y2.
Ces règles peuvent être remplacées par les productions associées au non-terminal cible pour simplifier la grammaire.

```
void supprimer_unite(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        int index = 0;

        // Parcourir les productions
        while (index < rule->production_count) {
            char *prod = rule->productions[index];

            // Vérifier si c'est une règle unité (une lettre
               majuscule suivie d'un chiffre)
            if (strlen(prod) > 1 && isupper(prod[0]) && isdigit(
               prod[1])) {
                char target_non_terminal[MAX_SYMBOLS];
                strcpy(target_non_terminal, prod);

                // Trouver la règle associée
                int found = 0;
                for (int j = 0; j < grammaire->rule_count; j++) {
                    if (strcmp(grammaire->rules[j].non_terminal,
                       target_non_terminal) == 0) {
                        Rule *target_rule = &grammaire->rules[j];
                        found = 1;

                        // Ajouter les productions de la règle
                           cible à la règle courante
                        for (int k = 0; k < target_rule->
                           production_count; k++) {
                            char *new_prod = target_rule->
                               productions[k];

                            // Vérifier si la production existe d
                               éjà
                            int exists = 0;
                            for (int l = 0; l < rule->
                               production_count; l++) {
```

```
                                if (strcmp(rule->productions[l],
                                    new_prod) == 0) {
                                    exists = 1;
                                    break;
                                }
                            }

                            // Ajouter la production si elle n'
                                existe pas encore
                            if (!exists && rule->production_count
                                < MAX_RULES) {
                                strcpy(rule->productions[rule->
                                    production_count++], new_prod)
                                    ;
                            }
                        }
                        break;
                    }
                }

                // Si la règle associée est trouvée, supprimer la
                    règle unité
                if (found) {
                    for (int k = index; k < rule->
                        production_count - 1; k++) {
                        strcpy(rule->productions[k], rule->
                            productions[k + 1]);
                    }
                    rule->production_count--;
                } else {
                    index++; // Passer à la production suivante
                        si aucune règle associée
                }
            } else {
                index++; // Passer à la production suivante si ce
                    n'est pas une règle unité
            }
        }
    }
}
```

3.4 fonction pour supprimer les non terminaux qui sont en tete

La fonction supprimer_non_terminaux_en_tete vise à remplacer les non-terminaux en tête de production (ceux qui apparaissent en premier dans une production) par leurs productions associées. Cela permet de "dérouler" les non-terminaux en tête pour simplifier la grammaire.

```
void supprimer_non_terminaux_en_tete(Grammaire *grammaire) {
    int changes;
```

```
do {
    changes = 0; // Indicateur de modifications

    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        for (int j = 0; j < rule->production_count; j++) {
            char *prod = rule->productions[j];

            // Identifier si le premier symbole est un non-
               terminal valide (Majuscule + Chiffre
               uniquement)
            if (strlen(prod) > 1 && isupper(prod[0]) &&
               isdigit(prod[1])) {
                char non_terminal_tete[MAX_SYMBOLS] = {0};

                // Extraire le non-terminal (ex: "A0")
                snprintf(non_terminal_tete, 3, "%c%c", prod
                   [0], prod[1]);

                // Vérifier si ce non-terminal existe dans
                   les règles
                int found = 0;
                for (int l = 0; l < grammaire->rule_count; l
                   ++) {
                    if (strcmp(grammaire->rules[l].
                       non_terminal, non_terminal_tete) == 0)
                        {
                        found = 1;
                        Rule *target_rule = &grammaire->rules
                           [l];

                        // Remplacer le non-terminal en tête
                           par ses productions
                        for (int m = 0; m < target_rule->
                           production_count; m++) {
                            char nouvelle_production[
                               MAX_SYMBOLS];

                            // Construire la nouvelle
                               production
                            snprintf(nouvelle_production,
                               sizeof(nouvelle_production), "
                               %s%s",
                                     target_rule->productions
                                        [m], prod + 2);

                            // Vérifier si cette nouvelle
                               production existe déjà
                            int existe = 0;
```

```
                                for (int n = 0; n < rule->
                                    production_count; n++) {
                                    if (strcmp(rule->productions[
                                        n], nouvelle_production)
                                        == 0) {
                                        existe = 1;
                                        break;
                                    }
                                }

                                // Ajouter la nouvelle production
                                    si elle n'existe pas
                                if (!existe && rule->
                                    production_count < MAX_RULES)
                                    {
                                    strcpy(rule->productions[rule
                                        ->production_count++],
                                        nouvelle_production);
                                }
                            }

                            // Supprimer l'ancienne production
                            for (int m = j; m < rule->
                                production_count - 1; m++) {
                                strcpy(rule->productions[m], rule
                                    ->productions[m + 1]);
                            }
                            rule->production_count--;
                            j--; // Réexaminer la position
                                actuelle après suppression

                            changes = 1; // Indiquer qu'une
                                modification a été effectuée
                            break;
                        }
                    }

                    // Si le non-terminal n'existe pas dans les r
                        ègles, ce n'est pas une erreur ici
                    if (!found) {
                        continue;
                    }
                }
            }
        }
    } while (changes); // Répéter jusqu'à ce qu'il n'y ait plus
        de modifications
}
```

3.5 fonction pour supprimer les terminaux qui ne sont pas en tete

La fonction supprimer_terminaux_non_en_tete est conçue pour remplacer les terminaux non présents en tête des productions par de nouveaux non-terminaux.

```
void supprimer_terminaux_non_en_tete(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        for (int j = 0; j < rule->production_count; j++) {
            char *prod = rule->productions[j];
            char new_production[MAX_SYMBOLS] = "";
            int changed = 0;

            // Vérifier chaque caractère dans la production
            for (int k = 0; prod[k] != '\0'; k++) {
                if (islower(prod[k]) && k != 0) { // Si un
                    terminal n'est pas en tête
                    // Créer un nouveau non-terminal pour ce
                        terminal
                    char new_non_terminal[MAX_SYMBOLS];
                    generate_non_terminal(new_non_terminal,
                        grammaire);

                    // Ajouter une nouvelle règle pour ce
                        terminal
                    Rule new_rule;
                    strcpy(new_rule.non_terminal,
                        new_non_terminal);
                    new_rule.production_count = 1;
                    snprintf(new_rule.productions[0], MAX_SYMBOLS
                        , "%c", prod[k]);
                    grammaire->rules[grammaire->rule_count++] =
                        new_rule;

                    // Remplacer le terminal par le nouveau non-
                        terminal
                    snprintf(new_production + strlen(
                        new_production), MAX_SYMBOLS - strlen(
                        new_production), "%s", new_non_terminal);
                    changed = 1;
                } else {
                    // Ajouter le caractère original s'il n'est
                        pas modifié
                    snprintf(new_production + strlen(
                        new_production), MAX_SYMBOLS - strlen(
                        new_production), "%c", prod[k]);
                }
            }

            // Mettre à jour la production si des changements ont
                été effectués
```

```
            if (changed) {
                strcpy(rule->productions[j], new_production);
            }
        }
    }
}
```

3.6 suppression des regles avec plus de 2 non terminaux dans le membre droit

3.6.1 non_terminal_in_rule

La fonction non_terminal_in_rule est utilisée pour vérifier si un non-terminal donné apparaît dans une production de règle.

```
int non_terminal_in_rule(const Grammaire *grammaire, const char *
   non_terminal, const char *rule_production) {
    int len = strlen(rule_production);

    for (int i = 0; i < len; i += 2) {
        if (isupper(rule_production[i]) && isdigit(
           rule_production[i + 1])) {
            char current_non_terminal[MAX_SYMBOLS];
            snprintf(current_non_terminal, 3, "%c%c",
               rule_production[i], rule_production[i + 1]);
            if (strcmp(current_non_terminal, non_terminal) == 0)
               {
                return 1; // Le non-terminal est trouvé dans la
                   production
            }
        }
    }
    return 0; // Non-terminal absent de la production
}
```

3.6.2 la fonction pour supprimer les regles qui contiennent plus de 2 non terminaux

La fonction supprimer_regles_avec_plus_de_deux_non_terminaux est utilisée pour décomposer les règles ayant plus de deux non-terminaux en plusieurs règles en creant de nouveau non terminaux .

```
void supprimer_regles_avec_plus_de_deux_non_terminaux(Grammaire *
   grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        for (int j = 0; j < rule->production_count; j++) {
            char *prod = rule->productions[j];
            int len = strlen(prod);
```

```
// Compter les non-terminaux dans la production
int non_terminal_count = 0;
for (int k = 0; k < len; k += 2) {
    if (isupper(prod[k]) && isdigit(prod[k + 1])) {
        non_terminal_count++;
    }
}

// Si plus de deux non-terminaux, procéder à la dé
   composition
if (non_terminal_count > 2) {
    char current_prod[MAX_SYMBOLS];
    strcpy(current_prod, prod); // Copie la
       production actuelle

    char new_non_terminal[MAX_SYMBOLS];
    char remaining_prod[MAX_SYMBOLS];

    // Initialisation : traiter le premier non-
       terminal
    strncpy(remaining_prod, current_prod + 2,
       MAX_SYMBOLS - 2);
    remaining_prod[MAX_SYMBOLS - 2] = '\0';

    do {
        generate_non_terminal(new_non_terminal,
           grammaire);
    } while (non_terminal_exists(grammaire,
       new_non_terminal) ||
             non_terminal_in_rule(grammaire,
                new_non_terminal, prod));

    snprintf(rule->productions[j], MAX_SYMBOLS, "%.*s
       %s", 2, current_prod, new_non_terminal);

    // Créer de nouvelles règles pour gérer le reste
    while (strlen(remaining_prod) > 2) {
        char first_non_terminal[MAX_SYMBOLS];
        strncpy(first_non_terminal, remaining_prod,
           2);
        first_non_terminal[2] = '\0';

        // Générer un nouveau non-terminal
        char temp_non_terminal[MAX_SYMBOLS];
        do {
            generate_non_terminal(temp_non_terminal,
               grammaire);
        } while (non_terminal_exists(grammaire,
           temp_non_terminal) ||
                 non_terminal_in_rule(grammaire,
                    temp_non_terminal, prod));
```

```
                    Rule new_rule;
                    strcpy(new_rule.non_terminal,
                       new_non_terminal);
                    snprintf(new_rule.productions[0], MAX_SYMBOLS
                       , "%s%s", first_non_terminal,
                       temp_non_terminal);
                    new_rule.production_count = 1;
                    grammaire->rules[grammaire->rule_count++] =
                       new_rule;

                    strcpy(new_non_terminal, temp_non_terminal);
                    memmove(remaining_prod, remaining_prod + 2,
                       strlen(remaining_prod) - 1);
                }

                // Cas où il reste exactement deux non-terminaux
                if (strlen(remaining_prod) == 2) {
                    Rule final_rule;
                    strcpy(final_rule.non_terminal,
                       new_non_terminal);
                    snprintf(final_rule.productions[0],
                       MAX_SYMBOLS, "%s", remaining_prod);
                    final_rule.production_count = 1;
                    grammaire->rules[grammaire->rule_count++] =
                       final_rule;
                }
            }
        }
    }
}
```

3.7 suppression de la recursivite gauche immediate et plus complexe

La fonction supprimer_recursivite_gauche transforme une regle de la forme de A0 → A0 | en A0 → A1 et elle cree: Aa → Aa | E

```
void supprimer_recursivite_gauche(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule_i = &grammaire->rules[i];

        // Identifier les productions récursives et non ré
           cursives
        char recursive_productions[MAX_RULES][MAX_SYMBOLS] = {0};
        char non_recursive_productions[MAX_RULES][MAX_SYMBOLS] =
           {0};
        int recursive_count = 0, non_recursive_count = 0;

        for (int j = 0; j < rule_i->production_count; j++) {
            if (strncmp(rule_i->productions[j], rule_i->
               non_terminal, strlen(rule_i->non_terminal)) == 0)
```

```
            {
                // Production récursive
                strcpy(recursive_productions[recursive_count++],
                    rule_i->productions[j] + strlen(rule_i->
                    non_terminal));
            } else {
                // Production non récursive
                strcpy(non_recursive_productions[
                    non_recursive_count++], rule_i->productions[j
                    ]);
            }
        }

        // Si aucune récursivité gauche, passer à la règle
            suivante
        if (recursive_count == 0) {
            continue;
        }

        // Générer un nouveau non-terminal pour gérer la ré
            cursivité
        char new_non_terminal[MAX_SYMBOLS];
        generate_non_terminal(new_non_terminal, grammaire);

        // Remplacer les règles de rule_i avec les productions
            non récursives suivies du nouveau non-terminal
        rule_i->production_count = 0;
        for (int j = 0; j < non_recursive_count; j++) {
            snprintf(rule_i->productions[rule_i->production_count
                ++], MAX_SYMBOLS, "%s%s",
                    non_recursive_productions[j],
                        new_non_terminal);
        }

        // Ajouter les règles pour le nouveau non-terminal
        Rule new_rule;
        strcpy(new_rule.non_terminal, new_non_terminal);
        new_rule.production_count = 0;
        for (int j = 0; j < recursive_count; j++) {
            snprintf(new_rule.productions[new_rule.
                production_count++], MAX_SYMBOLS, "%s%s",
                    recursive_productions[j], new_non_terminal);
        }
        strcpy(new_rule.productions[new_rule.production_count++],
            "E"); // Ajout de epsilon

        // Ajouter le nouveau non-terminal à la grammaire
        grammaire->rules[grammaire->rule_count++] = new_rule;

    }
```

```
}
```

3.8 suppression de l'axiome de membre droit des regles

La fonction ajouter_regle_pour_axe introduit un nouveau non terminal et remplace toutes les occurrences de l'axiome par ce non terminal et ensuite cree une nouvelle regle de l'axiome vers ce nouveau non terminal. on a choisi de remettre la regle de l'axiome vers le nouveau non terminal au debut de la grammaire

```
void ajouter_regle_pour_axe(const char *axiome, Grammaire *
   grammaire) {
    // Vérifier si l'axiome est présent dans les membres droits
    int axiome_present = 0;

    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];
        for (int j = 0; j < rule->production_count; j++) {
            if (strstr(rule->productions[j], axiome)) {
                axiome_present = 1;
                break;
            }
        }
        if (axiome_present) break; // Sortir dès que l'axiome est
            trouvé
    }

    // Si l'axiome n'est pas présent dans les membres droits, ne
       rien modifier
    if (!axiome_present) {
        printf("Aucune␣règle␣ne␣contient␣l'axiome␣'%s'␣dans␣ses␣
           membres␣droits.␣Pas␣de␣modification␣nécessaire.\n",
           axiome);
        return;
    }

    // Générer un nouveau non-terminal
    char nouveau_non_terminal[MAX_SYMBOLS];
    generate_non_terminal(nouveau_non_terminal, grammaire);

    // Ajouter une règle qui lie l'axiome au nouveau non-terminal
    Rule nouvelle_regle;
    strcpy(nouvelle_regle.non_terminal, axiome);
    strcpy(nouvelle_regle.productions[0], nouveau_non_terminal);
    nouvelle_regle.production_count = 1;

    if (grammaire->rule_count < MAX_RULES) {
        // Déplacer toutes les règles existantes vers la droite
           pour insérer la nouvelle règle en première position
        for (int i = grammaire->rule_count; i > 0; i--) {
            grammaire->rules[i] = grammaire->rules[i - 1];
        }
```

```
        // Ajouter la nouvelle règle en première position
        grammaire->rules[0] = nouvelle_regle;
        grammaire->rule_count++;
    } else {
        fprintf(stderr, "Erreur : Limite de règles atteinte, 
            impossible d'ajouter la nouvelle règle.\n");
        return;
    }

    // Parcourir toutes les règles pour remplacer les occurrences
        de l'axiome par le nouveau non-terminal
    for (int i = 1; i < grammaire->rule_count; i++) { // Commence
        à 1 pour ignorer la règle ajoutée
        Rule *rule = &grammaire->rules[i];

        // Si le non-terminal de la règle est l'axiome, le
            remplacer par le nouveau non-terminal
        if (strcmp(rule->non_terminal, axiome) == 0) {
            strcpy(rule->non_terminal, nouveau_non_terminal);
        }

        // Parcourir les productions et remplacer chaque
            occurrence de l'axiome
        for (int j = 0; j < rule->production_count; j++) {
            char *production = rule->productions[j];
            char temp[MAX_SYMBOLS] = "";
            char *pos = production;

            // Remplacer toutes les occurrences de l'axiome par
                le nouveau non-terminal
            while ((pos = strstr(pos, axiome)) != NULL) {
                // Copier la partie avant l'axiome
                strncat(temp, production, pos - production);
                strcat(temp, nouveau_non_terminal);
                production = pos + strlen(axiome);
                pos = production;
            }

            // Ajouter la partie restante
            strcat(temp, production);
            strcpy(rule->productions[j], temp);
        }
    }
}
```

3.9 supprimer les terminaux des regles qui sont de longueur au moins egale a 2

3.9.1 suppression des terminaux dans les regles de longueur au moins egale a 2

La fonction tranformRule transforme une règle individuelle d'une grammaire. Son but est de remplacer les terminaux dans les productions dont la longueur est supérieure

à 1 par des non-terminaux.

```
void transformRule(Rule *rule, Grammaire *grammaire) {
    for (int i = 0; i < rule->production_count; i++) {
        char *production = rule->productions[i];
        char nouvelle_production[MAX_SYMBOLS] = "";
        int len = strlen(production);

        for (int j = 0; j < len; j++) {
            if (isTerminal(production[j]) && len > 1) {
                // Remplacer tous les terminaux dans une
                   production de taille > 1
                char nouveau_non_terminal[MAX_SYMBOLS];
                generate_non_terminal(nouveau_non_terminal,
                   grammaire);

                // Créer une nouvelle règle associant le terminal
                    au non-terminal
                Rule nouvelle_regle;
                strcpy(nouvelle_regle.non_terminal,
                   nouveau_non_terminal);
                nouvelle_regle.production_count = 1;
                snprintf(nouvelle_regle.productions[0],
                   MAX_SYMBOLS, "%c", production[j]);

                // Ajouter la règle à la grammaire
                if (grammaire->rule_count < MAX_RULES) {
                    grammaire->rules[grammaire->rule_count++] =
                       nouvelle_regle;
                } else {
                    fprintf(stderr, "Erreur␣:␣Limite␣de␣règles␣
                       atteinte,␣impossible␣d'ajouter␣une␣
                       nouvelle␣règle.\n");
                    return;
                }

                // Remplacer le terminal par le nouveau non-
                   terminal dans la production
                strcat(nouvelle_production, nouveau_non_terminal)
                   ;
            } else {
                // Garder les non-terminaux ou les terminaux isol
                   és
                char temp[2] = {production[j], '\0'};
                strcat(nouvelle_production, temp);
            }
        }

        // Remplacer l'ancienne production par la nouvelle
        strcpy(rule->productions[i], nouvelle_production);
    }
}
```

3.9.2 suppression des terminaux dans dans toutes les regles longueur au moins egale a 2

```
void transform(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        transformRule(&grammaire->rules[i], grammaire);
    }
}
```

4 fonctions principales

4.1 transformation en forme normale de chomsky

apres avoir fait toutes les sousfonctions de Chomsky on doit les appeler dans un corps de fonction suivant un ordre precis, on a donc :

```
void transformer_en_chomsky(Grammaire *grammaire, const char *
   axiome) {
    printf("Début␣de␣la␣transformation␣en␣forme␣normale␣de␣
       Chomsky\n");

    printf("\nÉtape␣1␣:␣Supprimer␣la␣récursivité␣gauche\n");
    supprimer_recursivite_gauche(grammaire);
    afficher_grammaire(grammaire);

   supprimer_E_non_isole(grammaire);
   afficher_grammaire(grammaire);

   printf("\nÉtape␣2␣:␣factoriser\n");
   factoriser(grammaire);
   afficher_grammaire(grammaire);
    printf("Étape␣3␣:␣Retirer␣l'axiome␣des␣membres␣droits\n");
    ajouter_regle_pour_axe(axiome, grammaire);
    afficher_grammaire(grammaire);

    printf("\nÉtape␣4␣:␣Supprimer␣les␣terminaux␣dans␣le␣membre␣
       droit␣des␣règles␣de␣longueur␣au␣moins␣deux\n");
    transform(grammaire);
    afficher_grammaire(grammaire);



    printf("\nÉtape␣5␣:␣Supprimer␣les␣règles␣avec␣plus␣de␣deux␣
       non-terminaux\n");
    supprimer_regles_avec_plus_de_deux_non_terminaux(grammaire);
    afficher_grammaire(grammaire);

    printf("\nÉtape␣6␣:␣Supprimer␣les␣règles␣X␣   ␣  ␣sauf␣si␣X␣
       est␣l'axiome\n");
    supprimer_epsilon(grammaire, axiome);
    afficher_grammaire(grammaire);
```

```
    printf("\nÉtape 7 : Supprimer les règles unité X    Y\n");
    supprimer_unite(grammaire);
    afficher_grammaire(grammaire);

    printf("\nÉtape 8 : nettoyer la grammaire\n");
     regrouper_terminaux(grammaire);
    afficher_grammaire(grammaire);

    printf("Fin de la transformation en forme normale de Chomsky\
       n");
    afficher_grammaire(grammaire);
}
```

4.2 transformation en forme normale de Greibach

```
void greibach(Grammaire *grammaire, const char *axiome) {
    // Étape 0 : Factoriser les règles (simplification préalable)
    printf("==== Application de la factorisation ====\n");
    factoriser(grammaire);
    afficher_grammaire(grammaire);

    // Étape 2 : Supprimer la récursivité gauche
    printf("\n==== Suppression de la récursivité gauche ====\n");
    supprimer_recursivite_gauche(grammaire);
    afficher_grammaire(grammaire);

    supprimer_E_non_isole(grammaire);
   afficher_grammaire(grammaire);

    // Étape 3 : Ajouter une règle pour  l axiome
    printf("\n==== Ajout de la règle pour l'axiome ====\n");
    ajouter_regle_pour_axe((char *)axiome, grammaire);
    afficher_grammaire(grammaire);

    // Étape 4 : Supprimer les règles epsilon
    printf("\n==== Suppression des règles epsilon ====\n");
    supprimer_epsilon(grammaire, axiome);
    afficher_grammaire(grammaire);

    // Étape 5 : Supprimer les règles unité
    printf("\n==== Suppression des règles unité ====\n");
    supprimer_unite(grammaire);
    afficher_grammaire(grammaire);


    // Étape 6 : Supprimer les non-terminaux en tête
    printf("\n==== Suppression des non-terminaux en tête des rè
       gles ====\n");
    supprimer_non_terminaux_en_tete(grammaire);
```

```
    afficher_grammaire(grammaire);

    // Étape 7 : Supprimer les terminaux qui ne sont pas en tête
    printf("\n====␣Suppression␣des␣terminaux␣non␣en␣tête␣====\n")
       ;
    supprimer_terminaux_non_en_tete(grammaire);
    afficher_grammaire(grammaire);

    // Étape 8 : nettoyer la grammaire
    printf("\n====␣nettoyer␣la␣grammaire␣====\n");
    regrouper_terminaux(grammaire);
    printf("\nTransformation␣en␣forme␣normale␣de␣Greibach␣terminé
       e.\n");
    afficher_grammaire(grammaire);
}
```

4.3 verification si une grammaire est en forme normale de chomsky

elle doit etre de la forme : X->YZ X->a X->E si X est l'axiome

```
int isChomsky(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        for (int j = 0; j < rule->production_count; j++) {
            const char *production = rule->productions[j];
            printf("Vérification␣de␣la␣production␣:␣%s\n",
               production);

            // Cas 1 : Un seul terminal
            if (strlen(production) == 1 && isTerminal(production
               [0])) {
                printf("Valide␣:␣terminal␣isolé\n");
                continue;
            }

            // Cas 2 : Deux non-terminaux
            if (strlen(production) == 4) { // Longueur 4, ex : "
               Y8Z0"
                char left[3] = {production[0], production[1], '\0
                   '}; // Premier non-terminal : Y8
                char right[3] = {production[2], production[3], '
                   \0'}; // Second non-terminal : Z0

                if (isNonTerminal(left) && isNonTerminal(right))
                   {
                    printf("Valide␣:␣deux␣non-terminaux␣(%s␣et␣%s
                       )\n", left, right);
                    continue;
                } else {
                    if (!isNonTerminal(left)) {
```

```
                    printf("Non valide : %s n'est pas un non-
                        terminal valide.\n", left);
                }
                if (!isNonTerminal(right)) {
                    printf("Non valide : %s n'est pas un non-
                        terminal valide.\n", right);
                }
            }
        }

        // Cas 3 : Axiome produisant epsilon
        if (strcmp(production, "E") == 0 &&
            strcmp(rule->non_terminal, grammaire->rules[0].
                non_terminal) == 0) {
            printf("Valide : epsilon pour l'axiome\n");
            continue;
        }

        printf("Non valide : La production %s ne respecte pas
            CNF.\n", production);
        return 0;
    }
  }
  return 1; // Toutes les règles sont valides
}
```

4.4 verification si une grammaire est en forme normale de Greibach

elle doit etre de la forme : X->aA0A1...An X->a X->E si X est l'axiome

```
int isGreibach(Grammaire *grammaire) {
    for (int i = 0; i < grammaire->rule_count; i++) {
        Rule *rule = &grammaire->rules[i];

        for (int j = 0; j < rule->production_count; j++) {
            const char *production = rule->productions[j];
            printf("Vérification de la production : %s\n",
                production);

            // Cas 1 : Axiome produisant epsilon
            if (strcmp(production, "E") == 0 &&
                strcmp(rule->non_terminal, grammaire->rules[0].
                    non_terminal) == 0) {
                printf("Valide : epsilon pour l'axiome\n");
                continue;
            }

            // Cas 2 : La production commence par un terminal
            if (isTerminal(production[0])) {
                int valide = 1;
```

```
                // Vérifier que les symboles suivants sont des
                   non-terminaux
                for (int k = 1; k < strlen(production); k += 2) {
                    char symbol[3] = {production[k], production[k
                        + 1], '\0'};
                    if (!isNonTerminal(symbol)) {
                        valide = 0;
                        break;
                    }
                }

                if (valide) {
                    printf("Valide : commence par un terminal
                       suivi de non-terminaux\n");
                    continue;
                }
            }

            // Si aucune condition n'est remplie, la production n
               'est pas valide
            printf("Non valide : La production %s ne respecte pas
               GNF.\n", production);
            return 0;
        }
    }
    return 1; // Toutes les productions respectent GNF
}
```

5 Exécution du programme

Voici des exemples d'entrée et de sortie.

5.1 Exemple 1 d'entrée

Listing 6: Exemple d'entrée

```
Grammaire:
S -> aSb
S -> E
```

5.2 Exemple 1 de sortie

Listing 7: Exemple de sortie

```
Grammaire originale :
Grammaire:
S -> aSb
S -> E
Avant réécriture:
Grammaire:
S -> aSb
```

```
S -> E

Après réécriture:
Grammaire:
S -> aSb | E

==== Transformation en forme normale de Greibach ====
==== Application de la factorisation ====
Grammaire:
S -> aSb | E

==== Suppression de la récursivité gauche ====
Grammaire:
S -> aSb | E

==== Ajout de la règle pour l'axiome ====
Début : letter_index=25, number_index=9
Généré : Z9
Fin : letter_index=25, number_index=8
Grammaire:
S -> Z9
Z9 -> aZ9b | E

==== Suppression des règles epsilon ====
Grammaire:
S -> Z9 | E
Z9 -> aZ9b | ab

==== Suppression des règles unité ====
Grammaire:
S -> E | aZ9b | ab
Z9 -> aZ9b | ab

==== Suppression des non-terminaux en tête des règles ====
Grammaire:
S -> E | aZ9b | ab
Z9 -> aZ9b | ab

==== Suppression des terminaux non en tête ====
Début : letter_index=25, number_index=8
Généré : Z8
Fin : letter_index=25, number_index=7
Début : letter_index=25, number_index=7
Généré : Z7
Fin : letter_index=25, number_index=6
Début : letter_index=25, number_index=6
Généré : Z6
Fin : letter_index=25, number_index=5
Début : letter_index=25, number_index=5
Généré : Z5
Fin : letter_index=25, number_index=4
```

```
Grammaire:
S -> E | aZ9Z8 | aZ7
Z9 -> aZ9Z6 | aZ5
Z8 -> b
Z7 -> b
Z6 -> b
Z5 -> b

==== nettoyer la grammaire ====
Début : letter_index=25, number_index=4
Généré : Z4
Fin : letter_index=25, number_index=3

Transformation en forme normale de Greibach terminée.
Grammaire:
S -> E | aZ9Z4 | aZ4
Z9 -> aZ9Z4 | aZ4
Z4 -> b
Grammaire sauvegardée dans le fichier 'exemple.Transforme.
   greibach'.
Vérification de la production : E
Valide : epsilon pour l'axiome
Vérification de la production : aZ9Z4
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ4
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ9Z4
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ4
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
La grammaire est sous forme normale de Greibach.

==== Transformation en forme normale de Chomsky ====
Début de la transformation en forme normale de Chomsky

Étape 1 : Supprimer la récursivité gauche
Grammaire:
S -> aSb | E

Étape 2 : factoriser
Grammaire:
S -> aSb | E
Étape 3 : Retirer l'axiome des membres droits
Début : letter_index=25, number_index=3
Généré : Z3
Fin : letter_index=25, number_index=2
Grammaire:
S -> Z3
Z3 -> aZ3b | E
```

```
Étape 4 : Supprimer les terminaux dans le membre droit des règles
    de longueur au moins deux
Début : letter_index=25, number_index=2
Généré : Z2
Fin : letter_index=25, number_index=1
Début : letter_index=25, number_index=1
Généré : Z1
Fin : letter_index=25, number_index=0
Grammaire:
S -> Z3
Z3 -> Z2Z3Z1 | E
Z2 -> a
Z1 -> b

Étape 5 : Supprimer les règles avec plus de deux non-terminaux
Début : letter_index=25, number_index=0
Généré : Z0
Fin : letter_index=24, number_index=9
Début : letter_index=24, number_index=9
Généré : Y9
Fin : letter_index=24, number_index=8
Grammaire:
S -> Z3
Z3 -> Z2Z0 | E
Z2 -> a
Z1 -> b
Z0 -> Z3Y9
Y9 -> Z1

Étape 6 : Supprimer les règles X        sauf si X est l'axiome
Grammaire:
S -> Z3 | E
Z3 -> Z2Z0
Z2 -> a
Z1 -> b
Z0 -> Z3Y9 | Y9
Y9 -> Z1

Étape 7 : Supprimer les règles unité X     Y
Grammaire:
S -> E | Z2Z0
Z3 -> Z2Z0
Z2 -> a
Z1 -> b
Z0 -> Z3Y9 | b
Y9 -> b

Étape 8 : nettoyer la grammaire
Début : letter_index=24, number_index=8
Généré : Y8
```

```
Fin : letter_index=24, number_index=7
Début : letter_index=24, number_index=7
Généré : Y7
Fin : letter_index=24, number_index=6
Grammaire:
S -> E | Y8Z0
Z3 -> Y8Z0
Z0 -> Z3Y7 | b
Y8 -> a
Y7 -> b
Fin de la transformation en forme normale de Chomsky
Grammaire:
S -> E | Y8Z0
Z3 -> Y8Z0
Z0 -> Z3Y7 | b
Y8 -> a
Y7 -> b
Grammaire sauvegardée dans le fichier 'exemple.Transforme.chomsky
   '.
Vérification de la production : E
Valide : epsilon pour l'axiome
Vérification de la production : Y8Z0
Valide : deux non-terminaux (Y8 et Z0)
Vérification de la production : Y8Z0
Valide : deux non-terminaux (Y8 et Z0)
Vérification de la production : Z3Y7
Valide : deux non-terminaux (Z3 et Y7)
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : b
Valide : terminal isolé
La grammaire est en forme de Chomsky.
```

5.3 Exemple 2 d'entrée

Listing 8: Exemple d'entrée

```
Grammaire:
A0 -> A0a
A0 -> E
```

5.4 Exemple 2 de sortie

Listing 9: Exemple de sortie

```
Grammaire originale :
Grammaire:
A0 -> A0a
A0 -> E
Avant réécriture:
```

```
Grammaire:
A0 -> A0a
A0 -> E

Après réécriture:
Grammaire:
A0 -> A0a | E

==== Transformation en forme normale de Greibach ====
==== Application de la factorisation ====
Grammaire:
A0 -> A0a | E

==== Suppression de la récursivité gauche ====
Début : letter_index=25, number_index=9
Généré : Z9
Fin : letter_index=25, number_index=8
Grammaire:
A0 -> EZ9
Z9 -> aZ9 | E
Avant suppression de E : EZ9
Après suppression de E : Z9
Grammaire:
A0 -> Z9
Z9 -> aZ9 | E

==== Ajout de la règle pour l'axiome ====
Aucune règle ne contient l'axiome 'A0' dans ses membres droits.
   Pas de modification nécessaire.
Grammaire:
A0 -> Z9
Z9 -> aZ9 | E

==== Suppression des règles epsilon ====
Grammaire:
A0 -> Z9 | E
Z9 -> aZ9 | a

==== Suppression des règles unité ====
Grammaire:
A0 -> E | aZ9 | a
Z9 -> aZ9 | a

==== Suppression des non-terminaux en tête des règles ====
Grammaire:
A0 -> E | aZ9 | a
Z9 -> aZ9 | a

==== Suppression des terminaux non en tête ====
Grammaire:
A0 -> E | aZ9 | a
```

```
Z9 -> aZ9 | a

==== nettoyer la grammaire ====

Transformation en forme normale de Greibach terminée.
Grammaire:
A0 -> E | aZ9 | a
Z9 -> aZ9 | a
Grammaire sauvegardée dans le fichier 'exemple.Transforme.
   greibach'.
Vérification de la production : E
Valide : epsilon pour l'axiome
Vérification de la production : aZ9
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : a
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ9
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : a
Valide : commence par un terminal suivi de non-terminaux
La grammaire est sous forme normale de Greibach.

==== Transformation en forme normale de Chomsky ====
Début de la transformation en forme normale de Chomsky

Étape 1 : Supprimer la récursivité gauche
Début : letter_index=25, number_index=8
Généré : Z8
Fin : letter_index=25, number_index=7
Grammaire:
A0 -> EZ8
Z8 -> aZ8 | E
Grammaire:
A0 -> EZ8
Z8 -> aZ8 | E
Avant suppression de E : EZ8
Après suppression de E : Z8

Étape 2 : factoriser
Grammaire:
A0 -> Z8
Z8 -> aZ8 | E
Étape 3 : Retirer l'axiome des membres droits
Aucune règle ne contient l'axiome 'A0' dans ses membres droits.
   Pas de modification nécessaire.
Grammaire:
A0 -> Z8
Z8 -> aZ8 | E

Étape 4 : Supprimer les terminaux dans le membre droit des règles
    de longueur au moins deux
```

```
Début : letter_index=25, number_index=7
Généré : Z7
Fin : letter_index=25, number_index=6
Grammaire:
A0 -> Z8
Z8 -> Z7Z8 | E
Z7 -> a

Étape 5 : Supprimer les règles avec plus de deux non-terminaux
Grammaire:
A0 -> Z8
Z8 -> Z7Z8 | E
Z7 -> a

Étape 6 : Supprimer les règles X        sauf si X est l'axiome
Grammaire:
A0 -> Z8 | E
Z8 -> Z7Z8 | Z7
Z7 -> a

Étape 7 : Supprimer les règles unité X     Y
Grammaire:
A0 -> E | Z7Z8 | a
Z8 -> Z7Z8 | a
Z7 -> a

Étape 8 : nettoyer la grammaire
Début : letter_index=25, number_index=6
Généré : Z6
Fin : letter_index=25, number_index=5
Grammaire:
A0 -> E | Z6Z8 | a
Z8 -> Z6Z8 | a
Z6 -> a
Fin de la transformation en forme normale de Chomsky
Grammaire:
A0 -> E | Z6Z8 | a
Z8 -> Z6Z8 | a
Z6 -> a
Grammaire sauvegardée dans le fichier 'exemple.Transforme.chomsky
   '.
Vérification de la production : E
Valide : epsilon pour l'axiome
Vérification de la production : Z6Z8
Valide : deux non-terminaux (Z6 et Z8)
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : Z6Z8
Valide : deux non-terminaux (Z6 et Z8)
Vérification de la production : a
Valide : terminal isolé
```

```
Vérification de la production : a
Valide : terminal isolé
La grammaire est en forme de Chomsky.
```

5.5 Exemple 3 d'entrée

Listing 10: Exemple d'entrée

```
Grammaire:
S0 -> A0
S0 -> B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bB0
B0 -> C0bC0
C0 -> aC0bC0
C0 -> bC0aC0
C0 -> E
```

5.6 Exemple 3 de sortie

Listing 11: Exemple de sortie

```
Grammaire originale :
Grammaire:
S0 -> A0
S0 -> B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bB0
B0 -> C0bC0
C0 -> aC0bC0
C0 -> bC0aC0
C0 -> E
Avant réécriture:
Grammaire:
S0 -> A0
S0 -> B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bB0
B0 -> C0bC0
C0 -> aC0bC0
C0 -> bC0aC0
C0 -> E

Après réécriture:
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bB0 | C0bC0
```

```
C0 -> aC0bC0 | bC0aC0

==== Transformation en forme normale de Greibach ====
==== Application de la factorisation ====
Début : letter_index=25, number_index=9
Généré : Z9
Fin : letter_index=25, number_index=8
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | C0

==== Suppression de la récursivité gauche ====
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | C0
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | C0

==== Ajout de la règle pour l'axiome ====
Aucune règle ne contient l'axiome 'S0' dans ses membres droits.
   Pas de modification nécessaire.
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | C0

==== Suppression des règles epsilon ====
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | C0
```

```
==== Suppression des règles unité ====
Grammaire:
S0 -> A0 | B0
 0  -> C0aA0
A0 -> C0aC0
B0 -> C0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | aC0bC0 | bC0aC0

==== Suppression des non-terminaux en tête des règles ====
Grammaire:
S0 -> A0 | B0
 0  -> aC0bC0aA0 | bC0aC0aA0
A0 -> aC0bC0aC0 | bC0aC0aC0
B0 -> aC0bC0bZ9 | bC0aC0bZ9
C0 -> aC0bC0 | bC0aC0
Z9 -> B0 | aC0bC0 | bC0aC0

==== Suppression des terminaux non en tête ====
Début : letter_index=25, number_index=8
Généré : Z8
Fin : letter_index=25, number_index=7
Début : letter_index=25, number_index=7
Généré : Z7
Fin : letter_index=25, number_index=6
Début : letter_index=25, number_index=6
Généré : Z6
Fin : letter_index=25, number_index=5
Début : letter_index=25, number_index=5
Généré : Z5
Fin : letter_index=25, number_index=4
Début : letter_index=25, number_index=4
Généré : Z4
Fin : letter_index=25, number_index=3
Début : letter_index=25, number_index=3
Généré : Z3
Fin : letter_index=25, number_index=2
Début : letter_index=25, number_index=2
Généré : Z2
Fin : letter_index=25, number_index=1
Début : letter_index=25, number_index=1
Généré : Z1
Fin : letter_index=25, number_index=0
Début : letter_index=25, number_index=0
Généré : Z0
Fin : letter_index=24, number_index=9
Début : letter_index=24, number_index=9
Généré : Y9
Fin : letter_index=24, number_index=8
Début : letter_index=24, number_index=8
Généré : Y8
```

```
Fin : letter_index=24, number_index=7
Début : letter_index=24, number_index=7
Généré : Y7
Fin : letter_index=24, number_index=6
Début : letter_index=24, number_index=6
Généré : Y6
Fin : letter_index=24, number_index=5
Début : letter_index=24, number_index=5
Généré : Y5
Fin : letter_index=24, number_index=4
Début : letter_index=24, number_index=4
Généré : Y4
Fin : letter_index=24, number_index=3
Début : letter_index=24, number_index=3
Généré : Y3
Fin : letter_index=24, number_index=2
Grammaire:
S0 -> A0 | B0
 0  -> aC0Z8C0Z7A0 | bC0Z6C0Z5A0
A0 -> aC0Z4C0Z3C0 | bC0Z2C0Z1C0
B0 -> aC0Z0C0Y9Z9 | bC0Y8C0Y7Z9
C0 -> aC0Y6C0 | bC0Y5C0
Z9 -> B0 | aC0Y4C0 | bC0Y3C0
Z8 -> b
Z7 -> a
Z6 -> a
Z5 -> a
Z4 -> b
Z3 -> a
Z2 -> a
Z1 -> a
Z0 -> b
Y9 -> b
Y8 -> a
Y7 -> b
Y6 -> b
Y5 -> a
Y4 -> b
Y3 -> a

==== nettoyer la grammaire ====
Début : letter_index=24, number_index=2
Généré : Y2
Fin : letter_index=24, number_index=1
Début : letter_index=24, number_index=1
Généré : Y1
Fin : letter_index=24, number_index=0
zsh: segmentation fault  ./factorisation
```

la grammaire est tres longue pour notre logique, mais nous avons pense a regler le segmentation fault en supprimant les regles et non terminaux inutilises , et liberant la memoire ou ils etaient stockes.

5.7 Exemple 4 d'entrée

Listing 12: Exemple d'entrée

```
Grammaire:
Grammaire:
S -> A1bB1
S -> C1
B1 -> A1A1
B1 -> A1C1
C1 -> b
C1 -> c
A1 -> a
A1 -> E
```

5.8 Exemple 4 de sortie

Listing 13: Exemple de sortie

```
Grammaire originale :
Grammaire:
S -> A1bB1
S -> C1
B1 -> A1A1
B1 -> A1C1
C1 -> b
C1 -> c
A1 -> a
A1 -> E
Avant réécriture:
Grammaire:
S -> A1bB1
S -> C1
B1 -> A1A1
B1 -> A1C1
C1 -> b
C1 -> c
A1 -> a
A1 -> E

Après réécriture:
Grammaire:
S -> A1bB1 | C1
B1 -> A1A1 | A1C1
C1 -> b | c
A1 -> a | E

==== Transformation en forme normale de Greibach ====
==== Application de la factorisation ====
Début : letter_index=25, number_index=9
Généré : Z9
Fin : letter_index=25, number_index=8
```

```
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z9
C1 -> b | c
A1 -> a | E
Z9 -> A1 | C1

==== Suppression de la récursivité gauche ====
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z9
C1 -> b | c
A1 -> a | E
Z9 -> A1 | C1

==== Suppression de E-non-isolé si il existe
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z9
C1 -> b | c
A1 -> a | E
Z9 -> A1 | C1

==== Ajout de la règle pour l'axiome ====
Aucune règle ne contient l'axiome 'S' dans ses membres droits.
   Pas de modification nécessaire.
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z9
C1 -> b | c
A1 -> a | E
Z9 -> A1 | C1

==== Suppression des règles epsilon ====
Grammaire:
S -> A1bB1 | C1 | A1b | bB1 | b
B1 -> A1Z9 | Z9 | A1
C1 -> b | c
A1 -> a
Z9 -> A1 | C1

==== Suppression des règles unité ====
Grammaire:
S -> A1bB1 | A1b | bB1 | b | c
B1 -> A1Z9 | a | b | c
C1 -> b | c
A1 -> a
Z9 -> a | b | c

==== Suppression des non-terminaux en tête des règles ====
Grammaire:
```

```
S -> bB1 | b | c | abB1 | ab
B1 -> a | b | c | aZ9
C1 -> b | c
A1 -> a
Z9 -> a | b | c

==== Suppression des terminaux non en tête ====
Début : letter_index=25, number_index=8
Généré : Z8
Fin : letter_index=25, number_index=7
Début : letter_index=25, number_index=7
Généré : Z7
Fin : letter_index=25, number_index=6
Grammaire:
S -> bB1 | b | c | aZ8B1 | aZ7
B1 -> a | b | c | aZ9
C1 -> b | c
A1 -> a
Z9 -> a | b | c
Z8 -> b
Z7 -> b

==== nettoyer la grammaire ====
Début : letter_index=25, number_index=6
Généré : Z6
Fin : letter_index=25, number_index=5
Début : letter_index=25, number_index=5
Généré : Z5
Fin : letter_index=25, number_index=4

Transformation en forme normale de Greibach terminée.
Grammaire:
S -> bB1 | b | c | aZ5B1 | aZ5
B1 -> a | b | c | aZ9
C1 -> b | c
Z9 -> a | b | c
Z6 -> a
Z5 -> b
Grammaire sauvegardée dans le fichier 'exemple.Transforme.
   greibach'.
Vérification de la production : bB1
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : c
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ5B1
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ5
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : a
```

```
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : c
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : aZ9
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : c
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : a
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : c
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : a
Valide : commence par un terminal suivi de non-terminaux
Vérification de la production : b
Valide : commence par un terminal suivi de non-terminaux
La grammaire est sous forme normale de Greibach.

==== Transformation en forme normale de Chomsky ====
Début de la transformation en forme normale de Chomsky

Étape 1 : Supprimer la récursivité gauche
Grammaire:
S -> A1bB1 | C1
B1 -> A1A1 | A1C1
C1 -> b | c
A1 -> a | E

==== Suppression de E-non-isolé si il existe
Grammaire:
S -> A1bB1 | C1
B1 -> A1A1 | A1C1
C1 -> b | c
A1 -> a | E

Étape 2 : factoriser
Début : letter_index=25, number_index=4
Généré : Z4
Fin : letter_index=25, number_index=3
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z4
C1 -> b | c
A1 -> a | E
Z4 -> A1 | C1
Étape 3 : Retirer l'axiome des membres droits
```

```
Aucune règle ne contient l'axiome 'S' dans ses membres droits.
   Pas de modification nécessaire.
Grammaire:
S -> A1bB1 | C1
B1 -> A1Z4
C1 -> b | c
A1 -> a | E
Z4 -> A1 | C1

Étape 4 : Supprimer les terminaux dans le membre droit des règles
    de longueur au moins deux
Début : letter_index=25, number_index=3
Généré : Z3
Fin : letter_index=25, number_index=2
Grammaire:
S -> A1Z3B1 | C1
B1 -> A1Z4
C1 -> b | c
A1 -> a | E
Z4 -> A1 | C1
Z3 -> b

Étape 5 : Supprimer les règles avec plus de deux non-terminaux
Début : letter_index=25, number_index=2
Généré : Z2
Fin : letter_index=25, number_index=1
Début : letter_index=25, number_index=1
Généré : Z1
Fin : letter_index=25, number_index=0
Grammaire:
S -> A1Z2 | C1
B1 -> A1Z4
C1 -> b | c
A1 -> a | E
Z4 -> A1 | C1
Z3 -> b
Z2 -> Z3Z1
Z1 -> B1

Étape 6 : Supprimer les règles X        sauf si X est l'axiome
Grammaire:
S -> A1Z2 | C1 | Z2
B1 -> A1Z4 | Z4 | A1
C1 -> b | c
A1 -> a
Z4 -> A1 | C1
Z3 -> b
Z2 -> Z3Z1 | Z3
Z1 -> B1

Étape 7 : Supprimer les règles unité X     Y
```

```
Grammaire:
S -> A1Z2 | b | c | Z3Z1
B1 -> A1Z4 | a | b | c
C1 -> b | c
A1 -> a
Z4 -> a | b | c
Z3 -> b
Z2 -> Z3Z1 | b
Z1 -> A1Z4 | a | b | c

Étape 8 : nettoyer la grammaire
Début : letter_index=25, number_index=0
Généré : Z0
Fin : letter_index=24, number_index=9
Début : letter_index=24, number_index=9
Généré : Y9
Fin : letter_index=24, number_index=8
Grammaire:
S -> Z0Z2 | b | c | Y9Z1
B1 -> Z0Z4 | a | b | c
C1 -> b | c
Z4 -> a | b | c
Z2 -> Y9Z1 | b
Z1 -> Z0Z4 | a | b | c
Z0 -> a
Y9 -> b
Fin de la transformation en forme normale de Chomsky
Grammaire:
S -> Z0Z2 | b | c | Y9Z1
B1 -> Z0Z4 | a | b | c
C1 -> b | c
Z4 -> a | b | c
Z2 -> Y9Z1 | b
Z1 -> Z0Z4 | a | b | c
Z0 -> a
Y9 -> b
Grammaire sauvegardée dans le fichier 'exemple.Transforme.chomsky
   '.
Vérification de la production : Z0Z2
Valide : deux non-terminaux (Z0 et Z2)
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : c
Valide : terminal isolé
Vérification de la production : Y9Z1
Valide : deux non-terminaux (Y9 et Z1)
Vérification de la production : Z0Z4
Valide : deux non-terminaux (Z0 et Z4)
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : b
```

```
Valide : terminal isolé
Vérification de la production : c
Valide : terminal isolé
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : c
Valide : terminal isolé
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : c
Valide : terminal isolé
Vérification de la production : Y9Z1
Valide : deux non-terminaux (Y9 et Z1)
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : Z0Z4
Valide : deux non-terminaux (Z0 et Z4)
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : b
Valide : terminal isolé
Vérification de la production : c
Valide : terminal isolé
Vérification de la production : a
Valide : terminal isolé
Vérification de la production : b
Valide : terminal isolé
La grammaire est en forme de Chomsky.
```

6 Main

notre fonction main initialise la grammaire et en fait 2 copies pour chomsky et greibach et appelle les 2 fonctions pour transformer les 2 copies en formes normales de chomsky et greibach et finalement elle teste si les transformations ont bien ete effectuees: exemple d'un main:

Listing 14: le main

```
int main() {
    Grammaire grammaire_originale;

    // Lire la grammaire depuis un fichier
    if (lire_grammaire(&grammaire_originale, "exemple.general.txt
       ") == -1) {
        fprintf(stderr, "Erreur : Impossible de lire la grammaire
           .\n");
        return -1;
    }
```

```
    const char *axiome = grammaire_originale.rules[0].
       non_terminal;

    // Affichage initial de la grammaire
    printf("Grammaire originale :\n");
    afficher_grammaire(&grammaire_originale);

    // Affichage avant réécriture
    printf("Avant réécriture:\n");
    afficher_grammaire(&grammaire_originale);

    // Réécrire la grammaire
    rewriter_grammaire(&grammaire_originale);

    // Affichage après réécriture
    printf("\nAprès réécriture:\n");
    afficher_grammaire(&grammaire_originale);

    // Créer une copie de la grammaire originale pour chaque
       transformation
    Grammaire grammaire_greibach = grammaire_originale;
    Grammaire grammaire_chomsky = grammaire_originale;

    // Transformation en forme normale de Greibach
    printf("\n==== Transformation en forme normale de Greibach
       ====\n");
    greibach(&grammaire_greibach, axiome);
    sauvegarder_grammaire(&grammaire_greibach, "exemple.
       Transforme", 'g');
    if (isGreibach(&grammaire_greibach)) {
        printf("La grammaire est sous forme normale de Greibach.\
           n");
    } else {
        printf("La grammaire n'est PAS sous forme normale de
           Greibach.\n");
    }

    // Transformation en forme normale de Chomsky
    printf("\n==== Transformation en forme normale de Chomsky
       ====\n");
    transformer_en_chomsky(&grammaire_chomsky, axiome);
    sauvegarder_grammaire(&grammaire_chomsky, "exemple.Transforme
       ", 'c');
    if (isChomsky(&grammaire_chomsky)) {
        printf("La grammaire est en forme de Chomsky.\n");
    } else {
        printf("La grammaire n'est PAS en forme de Chomsky.\n");
    }

    return 0;
}
```

7 MakeFile

Voici une explication de ce que fait chaque commande de notre Makefile :

make
Compile le programme de la partie 1 :
Génère lex.yy.c à partir de grammaire.l avec flex.
Compile lex.yy.c et grammaire.c pour créer l'exécutable grammaire.

make run FILE=<fichier>.general.txt: par exemple : make run FILE=exemple2.general.txt
Exécute le programme grammaire avec un fichier d'entrée (exemple.general.txt).

make clean
Supprime les fichiers générés (l'exécutable grammaire et le fichier intermédiaire lex.yy.c) pour nettoyer ton répertoire.
2eme programme
Compilation du programme secondaire 'generates$_w ords'$: $makep2$
$Exécutionduprogrammesecondaire(generates_w ords)$: $makerunp2$
$voicimoncode$:

Programme secondaire 'generates$_w ords' P2_E XEC = generates_w ords P2_S RC = generate_w ords.c$
Compilateur CC = gcc CFLAGS = -lfl
Compilation par défaut pour 'grammaire' all: $(EXEC)$
Génération de lex.yy.c à partir de grammaire.l lex.yy.c: grammaire.l flex grammaire.l
Règle pour générer l'exécutable 'grammaire' $(EXEC)$:(SRC) $(CC) lex.yy.c grammaire.c -$
o(EXEC) $(CFLAGS)$
Commande pour exécuter le programme 'grammaire' run: $(EXEC)@echo"Usage : makerunFILE =$
$fichier > .general.txt"@if[-z"$(FILE)"]; then echo "Error: FILE non spécifié"; exit$
1; fi ./$(EXEC)$(FILE)
Commandes pour le programme 'generates$_w ords' p2 :(P2$_E XEC)$
$(P2_E XEC)$:(P2$_S RC$)(CC) -g $(P2_S RC) - o$(P2$_E XEC)$
runp2: $(P2_E XEC)$./(P2$_E XEC)$
Nettoyage des fichiers générés clean: rm -f $(EXEC) lex.yy.c$
cleanp2: rm -f $(P2_E XEC)$

8 Partie 2 : generation de mots de taille max n

on a cree un deuxieme programme pour generer tous les mots delongueur inferieure ou egale a n a partir des fichiers de grammaire de greibach ou chomsky. voici nos principales fonctions:

MAX_RULES, MAX_SYMBOLS, et MAX_WORD_LEN définissent les limites maximales pour le nombre de règles, de symboles, et la longueur d'un mot.

Les structures **Rule et Grammaire** servent à représenter une grammaire algébrique avec des règles et un axiome. dans Grammaire on ajoute aussi l'axiome.

la fonction **lire_grammaire** qui lit une grammaire depuis un fichier texte et la stocke dans la structure Grammaire.

la fonction **decomposer_mot**, le principe:
Décompose une chaîne en symboles, qui peuvent être des terminaux ou non-terminaux.
Fonctionnement : Si un caractère majuscule est suivi d'un chiffre (comme A1), il est traité comme un non-terminal. Sinon, chaque caractère est considéré comme un terminal.
la fonction **est_terminal** (identique a celle de l'ancien programme)
la fonction **trouver_productions** qui Récupère toutes les productions associées à un non-terminal donné. elle Parcourt les règles de la grammaire et compare le non-terminal demandé avec celui de chaque règle.
la fonction **generer_mots_recursif** Génère récursivement tous les mots possibles à partir de l'axiome.
Étapes : Décompose le mot courant en symboles. Vérifie si le mot courant est complètement terminal : Si oui, ajoute le mot à la liste des mots générés. Sinon, développe un non-terminal à la fois en le remplaçant par ses productions. Rappelle la fonction récursivement pour explorer toutes les possibilités.
la fonction **generer_mots** Génère, trie et enregistre les mots dans un fichier.
elle fait:
Utilise generer_mots_recursif pour produire les mots. Trie les mots générés par ordre lexicographique (d'abord par longueur, puis par ordre alphabétique). Écrit les mots triés dans le fichier de sortie .
la fonction **main** fait:
Lit une grammaire à partir d'un fichier spécifié (exemple.Transforme.chomsky).
Définit automatiquement l'axiome comme le premier non-terminal rencontré.
Génère les mots possibles jusqu'à une longueur maximale (3 dans cet exemple).
Sauvegarde les mots générés dans un fichier texte (mots_generes.txt).
TOUTES LES FONCTIONS SONT DIRECTEMENT IMPLEMENTEES DANS LE CODE

```
int main() {
    Grammaire grammaire_chomsky;
    Grammaire grammaire_greibach;

    // Charger la grammaire en forme normale de Chomsky
    if (lire_grammaire(&grammaire_chomsky, "exemple.Transforme.
       chomsky") == -1) {
        fprintf(stderr, "Erreur : Impossible de lire la grammaire en
            forme normale de Chomsky.\n");
        return -1;
    }
    // Définir automatiquement l'axiome comme le premier non-
       terminal pour Chomsky
    strcpy(grammaire_chomsky.axiome, grammaire_chomsky.rules[0].
       non_terminal);

    // Générer des mots pour la grammaire en forme normale de
       Chomsky
    printf("\n==== Génération de mots (Chomsky) ====");
    generer_mots(&grammaire_chomsky, 3, "mots_chomsky_generes.txt");

    // Charger la grammaire en forme normale de Greibach
    if (lire_grammaire(&grammaire_greibach, "exemple.Transforme.
       greibach") == -1) {
        fprintf(stderr, "Erreur : Impossible de lire la grammaire en
```

```
            forme normale de Greibach.\n");
        return -1;
    }
    // Définir automatiquement l'axiome comme le premier non-
       terminal pour Greibach
    strcpy(grammaire_greibach.axiome, grammaire_greibach.rules[0].
       non_terminal);

    // Générer des mots pour la grammaire en forme normale de
       Greibach
    printf("\n==== Génération de mots (Greibach) ====");
    generer_mots(&grammaire_greibach, 3, "mots_greibach_generes.txt
       ");

    return 0;
}
```

8.1 quelques exemples d'affichage:

8.1.1 exemple 1

voici un exemple complet avec la grammaire originale, les formes de chomsky et greibach et les mots generes pour les 2 formes:

```
Grammaire originale :
S : A1bB1
S :C1
B1 : A1A1
B1 :A1C1
C1: b
C1:c
A1 :a
A1 :E
Forme greibach :
S : bB1 | b | c | aZ5B1 | aZ5
B1 : a | b | c | aZ9
C1 : b | c
Z9 : a | b | c
Z6 : a
Z5 : b
Mots générés de longueur maximale 3   d apr ès  la forme greibach :
b
c
ab
ba
bb
bc
aba
abb
abc
baa
bab
```

```
bac
Forme chomsky :
S : Z0Z2 | b | c | Y9Z1
B1 : Z0Z4 | a | b | c
C1 : b | c
Z4 : a | b | c
Z2 : Y9Z1 | b
Z1 : Z0Z4 | a | b | c
Z0 : a
Y9 : b
Mots generes de longueur maximale 3 d'apres la forme chomsky
b
c
ab
ba
bb
bc
aba
abb
abc
baa
bab
bac
```

8.1.2 exemple 2

voici un exemple complet avec la grammaire originale, les formes de chomsky et greibach et les mots generes pour les 2 formes:

```
Grammaire originale :
A0 : A0a
A0 : E
Forme greibach :
A0 : E | aZ9 | a
Z9 : aZ9 | a
Mots générés de longueur maximale 3  d apr ès  la forme greibach :
E
a
aa
aaa
Forme chomsky :
A0 : E | Z6Z8 | a
Z8 : Z6Z8 | a
Z6 : a
Mots generes de longueur maximale 3 d'apres la forme chomsky
E
a
aa
aaa
```

8.1.3 exemple 3

voici un exemple complet avec la grammaire originale, les formes de chomsky et greibach et les mots generes pour les 2 formes:

```
Grammaire originale :
S : aSb
S : E
Forme greibach :
S : E | aZ9Z4 | aZ4
Z9 : aZ9Z4 | aZ4
Z4 : b
Mots générés de longueur maximale 3   d apr ès  la forme greibach :
E
ab
Forme chomsky :
S : E | Y8Z0
Z3 : Y8Z0
Z0 : Z3Y7 | b
Y8 : a
Y7 : b
Mots generes de longueur maximale 3 d'apres la forme chomsky
E
ab
```

8.1.4 exemple 4

voici un exemple complet avec la grammaire originale, les formes de chomsky et greibach et les mots generes pour les 2 formes:

```
Grammaire originale :
S : aSb
S : E
Forme greibach :
S : E | aZ9Z4 | aZ4
Z9 : aZ9Z4 | aZ4
Z4 : b
Mots générés de longueur maximale 4   d apr ès  la forme greibach :
E
ab
aabb
Forme chomsky :
S : E | Y8Z0
Z3 : Y8Z0
Z0 : Z3Y7 | b
Y8 : a
Y7 : b
Mots generes de longueur maximale 4 d'apres la forme chomsky
E
ab
aabb
```

9 conclusion

on a donc implemente tout ce qui a ete demande , cependant voici quelques points dont on voulait parler:
.**le fichier lex** nous avons cree un fichier lex et integre dans le makefile, cependant il ne sert pas a grand chose meme si on sait et on aurait pu en creer un qui peut reconnaitre nos tokens par exemple un non terminal : [A-Z 0-9] et un terminal [a-z] etc... mais le probleme est que nous avons commence par le code et la creation des fonctions et comme nous avions deja gere la lecture dans le fichier .c on ne se voyait pas reprendre depuis le debut, donc on a rendu un fichier lex mais ce n'etait pas ce qu'on voulait rendre .... .
.