I. Le cahier des charges

L'objectif du projet était de réaliser générateur d'analyse lexicale (Gal). Pour cela, plusieurs étapes sont nécessaires, et ont été découpées dans plusieurs fichiers :

1. lire le fichier d'entrée : lecture.h et lecture.c
2. construction de l'arbre de syntaxe abstraite : arbre.h et arbre.c
3. construction de l'automate fini non déterministe : nfa.h et nfa.c
4. construction de l'automate fini déterministe : dfa.h et dfa.c
5. construction de l'automate fini déterministe minimal : dfa_min.h et dfa_min.c TODO
6. génération du code source de l'analyseur lexicale TODO

II. Lecture du fichier et alphabet

Le fichier `alphabet.c` contient les fonctions de gestion de l'alphabet. Le but de ces fonctions est de pouvoir assez facilement changer d'alphabet en cas de besoin.

`char next_letter(char ch)` : Renvoie la première lettre de l'alphabet si `ch` = 0, ou la lettre suivant `ch` dans l'alphabet s'il y en a une et −1 s'il n'y en a plus.

`int letter_rank(char ch)` : Renvoie le rang de la lettre `ch` dans l'alphabet, ou −1 si ce n'est pas une lettre de l'alphabet.

Le fichier `lecture.c` contient les fonctions de lecture et de traitement du fichier.

`void lecture(char *nom, char *exp)` : Lit le fichier `nom` contenant une expression régulière, et stocke son contenu dans `exp`. Si le contenu du fichier ne finit pas par `\n` ou qu'il contient un caractère non reconnu, affiche un message d'erreur et termine le programme.

`void add_concat(char *src, char *dest)` : Ajoute des points à la chaîne `src` aux endroits des concaténations (par exemple `ab` devient `a.b`), remplace les couples de parenthèses vides () par ε (codé par _) et stocke le résultat dans `dest`.

`char *get_filename(char *fullpath)` : Prend en paramètre le chemin complet vers un fichier, et renvoie le nom du fichier en enlevant le chemin et l'extension.

III. Construction de l'arbre de syntaxe abstraite

1) Notation polonaise inversée

Pour construire l'arbre de syntaxe abstraite, après avoir un peu reformaté la chaîne grâce à la fonction `add_concat` décrite plus haut, j'ai choisi de passer par une écriture en notation polonaise inversée (NPI) en utilisant l'algorithme 1 (fonction `to_postfix`).

2) Construction de l'arbre

Ensuite, on construit l'arbre à partir de cette expression par l'algorithme 2 (fonction `to_tree`). Pour cela, un arbre (TREE) est un pointeur vers un nœud (NODE), lui même composé d'un caractère `val` indiquant le contenu du nœud, et de deux pointeurs `left` vers `right` vers les éventuels fils gauche et droit.
Pour ces deux étapes, on utilise une pile (implémentée dans `pile.c`) soit comme pile de caractères (avec les fonctions `push_char`, `pop_char` et `sommet_char` implémentée dans `pile.c`), soit comme pile d'arbres (avec `push_tree`, `pop_tree` et `sommet_tree` de `arbre.c`).

IV. Construction de l'automate fini non déterministe (NFA)

Entrée : Une chaîne de caractères `entry`
Sortie : Une chaîne de caractères `postfix` correspondant à la notation polonaise inversée de l'entrée
Traitement

```
Créer une pile vide
npi ← []
pour chaque caractère ch de entry faire
    si ch est une lettre alors
        ajouter ch à postfix
    sinon si ch est un parenthèse ouvrante alors
        empiler ch
    sinon si ch est une parenthèse fermante alors
        tant que pile est non vide et que le sommet de la pile n'est pas une parenthèse ouvrante faire
            dépiler un caractère et l'ajouter à postfix
        si pile est vide alors
            quitter // Problème de parenthésage
        sinon
            dépiler la parenthèse ouvrante
    sinon
        tant que pile est non vide et que le sommet de la pile a une priorité supérieure à ch faire
            dépiler un caractère et l'ajouter à postfix
        empiler ch
tant que pile est non vide faire
    dépiler un caractère et l'ajouter à postfix
    si le caractère est une parenthèse ouvrante alors
        quitter // Problème de parenthésage
```

Algorithme 1 : Algorithme de passage en notation polonaise inversée (Shunting-yard)

Entrées : Une chaîne de caractères `src` en NPI
Sortie : L'arbre `tree` correspondant
Traitement

```
si src est vide alors // langage vide
    Renvoyer un arbre vide
Créer une pile vide
pour chaque caractère ch de src faire
    Créer un nœud nd étiqueté par ch
    si ch est une lettre alors
        nd est une feuille
    sinon si ch = '*' alors// opérateur unaire
        dépiler le dernier arbre
        le mettre en fils gauche de nd
    sinon // opérateur binaire
        dépiler les deux derniers arbres
        le mettre comme fils gauche et droit de nd
    empiler nd
dépiler dans tree
si la pile n'est pas vide alors
    quitter // Expression mal construite
```

Algorithme 2 : Algorithme de création de l'arbre de syntaxe abstraite