PROJET COMPILATION– GÉNÉRATEUR D'ANALYSEUR LEXICAL

Delay Emmanuel

25 février 2020

Table des matières

1 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

On demandait de plus des représentations graphiques, qui sont créées dans le répertoire `pdf` en reprenant le nom du fichier passé en paramètre en lui ajoutant un suffixe `_tree` pour l'arbre de syntaxe abstraite, `_nfa` pour l'automate non déterministe, `_dfa` pour le déterministe et `_min` pour le déterministe minimal.

2 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.

3 Construction de l'arbre de syntaxe abstraite

3.1 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 (procédure `to_postfix`).

3.2 Construction de l'arbre

Ensuite, on construit l'arbre à partir de cette expression par l'algorithme 2 (procédure `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` et `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` implémentées dans `pile.c`), soit comme pile d'arbres (avec `push_tree`, `pop_tree` et `sommet_tree` de `arbre.c`).

3.3 Représentation graphique

La procédure `tree2file` commence par écrire l'en-tête d'un fichier qui pourra être converti en pdf grâce à `dot`. Ensuite, elle appelle une fonction récursive `tree2file_rec` qui ajoute les consignes de dessin de chaque nœud en lui associant un numéro qu'elle renvoie. Elle peut ainsi dessiner un arc vers les éventuels fils gauche et droit.

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)

4 Construction de l'automate fini non déterministe (NFA)

4.1 État d'un automate NFA

La plus grande difficulté pour moi a été de trouver une structure adaptée au problème. Au début, j'avais réfléchi en terme d'automate fini quelconque, et donc je pensais à une liste chaînée d'états, chaque état ayant une liste de successeurs. Mais je n'arrivais pas à voir comment appliquer l'algorithme du cours à cette structure.

En relisant ce dernier, j'ai fini par réagir que dans l'automate obtenu, chaque nœud avait au plus deux successeurs. De plus, le seul cas où il en avait deux était le cas où les deux correspondaient à une ε-transition et le seul cas où il n'y en a pas est le cas de l'état acceptant final. J'ai donc implémenté un état DFA par une structure STATE ayant 4 champs :

num : initialisé à NOT_VISITED. Le numéro de l'état sera attribué au moment de la représentation graphique, ce qui permettra d'éviter de traiter plusieurs fois le même nœud.

ch : un caractère définissant la nature de l'état. Si `ch` est une lettre de l'alphabet ou EPSILON, il n'y a qu'une arête sortante étiquetée par `ch`, si `ch` = SPLIT, il y a deux arêtes sortantes étiquetées par EPSILON, et si `ch` = ACCEPT alors il n'y a pas d'arête sortante et l'état est acceptant.

suiv et suiv2 : des pointeurs vers les éventuels successeurs.

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
        les mettre comme fils droit et gauche 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

4.2 Création du NFA

En implémentant un automate DFA comme une structure à deux champs : un pointeur `start` vers son état initial et un pointeur `end` vers son état final, on peut appliquer de façon complètement immédiate l'algorithme de McNaughton - Yamada - Thompson donné dans le cours. C'est ce que fait la fonction `tree2nfa`.

4.3 Représentation graphique

Là aussi, le procédure `nfa2file` commence par écrire l'en-tête avant d'appeler une procédure récursive `state2file` qui ajoute les consignes de dessin de chaque nœud en lui associant un numéro.

On utilise aussi le numéro pour repérer les états déjà traités : il ne faut traiter que ceux dont le numéro est encore NOT_VISITED.

Dans le cas du langage vide, l'état acceptant est isolé et ne sera alors pas atteint par `state2file`. On l'ajoute donc à la fin du fichier créé.

5 Construction de l'automate fini déterministe (DFA)

5.1 État d'un automate DFA

La difficulté ici était d'identifier un état du DFA grâce à un ensemble d'états de NFA. Pour ça, j'ai commencé par implémenter une structure LSTSTATES correspondant à une liste chaînée ordonnée d'états NFA. L'intérêt était de pouvoir assez rapidement (en temps linéaire par rapport au nombre d'états) ajouter un nouvel état sans répétition et comparer deux listes LSTSTATES pour savoir si elles sont égales ou pas. C'est ce que font les deux fonctions `add_state` et `cmp_lst_states`.

On peut alors représenter un état du DFA par une structure DSTATE ayant 5 champs :

lst_states : la liste ordonnée d'états du NFA décrite ci-dessus.

num : le numéro de l'état.

trans : un tableau de ALPHABET_LEN entiers donnant les numéros des états vers lesquels il existe une transition depuis l'état courant. L'indice du tableau correspond au rang dans l'alphabet de la lettre étiquetant la transition.

accept : indique si l'état est acceptant ou non. Ce champs est déterminé en regardant si la liste `lst_states` contient un état acceptant ou pas.

suiv : un pointeur vers l'état suivant à explorer dans l'application de l'algorithme du cours.

La fonction `new_dfa_state` permet de construire et renvoyer un tel état.

5.2 Création du DFA

Pour appliquer l'algorithme du cours, j'ai commencé par écrire une procédure récursive `eps_cloture_sin` déterminant l'ε-clôture d'un état `s` en l'ajoutant à une liste ordonnée d'états `cloture`. Le principe est d'essayer d'ajouter `s` à `cloture` et, s'il n'est pas déjà présent, à ajouter récursivement ses successeurs atteignables par ε-transition.

J'ai pu alors écrire une fonction `eps_cloture` prenant en paramètre une liste d'états et déterminant la réunion des ε-clôture de ses états en appelant `eps_cloture_single` pour chacun d'eux.

De même, la fonction `transition` parcours l'ensemble des états de LSTSTATES passée en paramètre, et les ajoute à une LSTSTATES `trans` si ils correspondent à une transition étiquetée par le caractère `ch`.

Enfin, une autre fonction utile pour appliquer l'algorithme du cours est la fonction `num_state` qui parcours l'ensemble des états DFA créés jusqu'à maintenant, et renvoie le numéro de l'état DFA correspondant à la LSTSTATES `lst` passée en paramètre, ou -1 si aucun état ne correspond.

L'application de l'algorithme du cours se fait alors presque mot à mot, en utilisant les fonctions définies dans `alphabet.h` pour parcourir l'ensemble des lettres de l'alphabet. La seule différence est que je commence par créer un état `puits` dont le successeur est l'état initial du DFA, c'est à dire l'ε-transition de l'état initial du NFA. L'intérêt de cet état sera décrit en 5.3.

5.3 Représentation graphique

Ici, la représentation graphique ne pose pas réellement de problème. Il suffit de parcourir la liste des états, et pour chacun le tableau de transitions. La seule précaution pour avoir un graphique lisible est de sauter toutes les transitions qui pointent vers l'état puits. Ça permet de ne pas faire apparaître les transitions étiquetées par des lettres qui ne sont pas concrètement utilisées dans l'expression régulière fournie en entrée. Le fait que l'état puits soit toujours l'état de numéro 0 simplifie cette étape.

6 Minimisation du DFA