Projet de programmation impérative

Rapport ; Automates LR1

Enzo De Carvalho Bittencourt

Projet 2021

Contents

1 Introduction

Le projet se porte sur l'implémentation d'un programme prenant un fichier .aut en argument représentant un automate LR1 et qui lit les lignes de l'entrée standard en indiquant pour chaque ligne lue si celle-ci appartient ou non au langage reconnu de l'automate.

Ici, sous les contraintes de l'énoncé, les automates ne travaillent que sur l'alphabet ASCII de 0 à 127 et fonctionnent à l'aide de quatre fonctions *action*, *décale*, *réduit* et *branchement* et d'une pile d'état, états ici limités au nombre de 256.

2 Implémentation

Les valeurs des fonctions *action*, *décale*, *réduit* et *branchement* ainsi que le nombre d'états de l'automate sont encodés dans le fichier .aut sous un format explicité par l'énoncé.

Par souci de standardisation le code, ses commentaires et le manuel `readme.md` sont entièrement rédigés en anglais. Il me semble pertinent d'expliciter certaines des traductions choisies dans ce projet:

accepte, *rejette*, *décale* et *réduit* ont été respectivement traduit en *accept*, *reject*, *shift* et *reduce*. Les fonction *action* et *branchement* sont traduites en *action* et *link*. Les mots *état* et *lettre* sont naturellement traduits en *state* et *letter*.

2.1 Lecture du fichier

Pour faciliter l'accès aux valeurs des fonctions plus tard dans le programme, j'ai commencé par écrire une procédure initiale qui lit et récupère les valeurs du fichier .aut et les stocke dans des matrices et listes. Ainsi, dans mon code

* *action(s,c)* est représentée par `action[s][c]`
* la première composante de *réduit(s)* par `reduce_n[s]`
* la seconde composante de *réduit(s)* par `reduce_letter[s]`
* *décales(s,c)* par `shift_matrix[s][c]`
* *branchement(s,a)* par `link_matrix[s][a]`

Cette procédure nommée `unpack` parcourt une seule fois toutes les lignes du fichier .aut (à l'aide des fonctions `fopen`, `fgets` et `fread`) et modifie durant sa lecture, par référence, ces tableaux et listes (ansi qu'un entier `n_state` représentant le nombre d'états de l'automate).

Toutes ces valeurs (à l'exception de `n_state` de type `int`) sont récupérées sous un type `char`, puisque elles sont directement encodées sur les octets du fichier.

De plus, on notera que `unpack` ne fait aucune gestion d'erreur lors de sa lecture (voir la partie Limitations).

2.2 Pile d'état

Les automates décrits dans l'énoncé fonctionnent à l'aide d'une pile d'état, systématiquement initialisé à l'état 0.

Pour l'implémentation de cette pile, j'ai donc crée une bibliothèque `stack.c` et son interface `stack.h` par souci de modularité (mon implémentation du programme est donc indépendant de mon implémentation des piles). Puisque les piles d'états des automates ne sont à priori pas limitées en taille, je les ai implémentées par des listes chaînées comme vu en cours.

Dans l'interface sont déclarés un type abstrait `stack` et 5 fonctions : une fonction `empty_stack` permettant de créer une pile vide, une fonction `is_empty_stack` permettant de vérifier si une pile est vide, une fonction `push_stack` et une autre `pop_stack` permettant respectivement d'empiler ou de dépiler un `char` dans la pile, et une dernière fonction `peek_stack` qui retourne le dernier `char` empilé dans la pile.

On notera également l'existence d'une procédure `print_debug_stack` non déclarée dans l'interface, mais présente dans le fichier `stack.c`, permettant d'afficher une visualisation d'une pile sur la sortie standard (cette procédure n'ayant servi que pour des tests le long de l'écriture de la bibliothèque.)

2.3 Fonctionnement global

Une fois que je pouvais récupérer correctement les données d'un fichier .aut et manipuler des piles, il ne manquait qu'à implémenter l'algorithme décrit dans le projet.

2.3.1 Déclarations préalables

Pour faciliter cette partie du travail j'ai préablement déclaré :

* un type `enum action` renommé en type `action`:

```
typedef enum action
{REJECT = 0, ACCEPT = 1, SHIFT = 2, REDUCE = 3}
action ;
```

* une fonction `action_func` qui retourne un objet de type `action`. Elle implémente la fonction *action* décrite dans l'énoncé : pour l'état courant `s` en dessus de la pile d'état et la lettre courante lue `c`, `action_func` indique quel comportement parmi `REJECT ACCEPT SHIFT REDUCE` doit suivre l'automate.

* une fonction `shift_func` qui implémente une partie du comportement de l'automate lorsque *action* vaut *Décale* en empilant l'état décrit par `shift_matrix` sur la pile d'état :

```
void shift_func(char state, char letter, char **shift_matrix, stack *s){
    char pushed_state = shift_matrix[(int) state][(int) letter];
    push_stack(pushed_state, s);
}
```

* une fonction `link_func` (fonctionnant de la même manière que `shift_func`) qui implémente une partie du comportement de l'automate lorsque *action* vaut *Réduit* en empilant l'état décrit par `link_matrix` sur la pile d'état.

Ces déclarations permettent une meilleures sémantique et modularité au sein du code, bien qu'elles ne gèrent pas l'entièreté des comportements de l'automate suivant la valeur de *action*.

On notera par exemple l'absence d'une possible fonction `reduce_func` qui aurait pu encoder le comportement de l'automate lorsque *action* vaut *Réduit* (en dépliant des états d'une pile passé par référence par exemple). Néanmoins, à ce moment de l'écriture de ce code, je ne savais pas si une telle modularité était possible plus tard.

2.3.2 Boucle `main()`

Au lancement du programme, celui-ci vérifie d'abord le nombres d'arguments donnés lors de son invocation. Sont déclaré ensuite plusieurs variables :

* l'entier `n_state` et les matrices de fonctions qui sont ensuite directement modifiées et remplies par la fonction `unpack` suivant le fichier .aut passé en argument au programme.

* Une pile d'état `state_stack` initialisée en pile vide.

* Un buffer `char *user_input` pour récupérer les lignes entrées par l'utilisateur.

* Un entier `DEBUG_FLAG` initialisé à 0. Si celui-ci vaut 1 dans le code, le programme affiche plus d'information lors de l'exécution utiles pour le débogage.

Le programme affiche ensuite sur la sortie standard un message assurant que le fichier a été correctement lu (comme demandé dans l'énoncé). Il rentre ensuite dans une boucle `while (1)` qui ne finit donc, à priori, jamais :

À chaque tour de cette boucle, le programme récupère le flux `stdin` dans `*user_input`, initialise la pile `state_stack` à l'état `'\0'`, déclare un entier `DONE_FLAG` initialisé à 0 qui sert d'invariant de boucle. Le programme rentre ensuite dans une seconde boucle `while (DONE_FLAG != 1)`.

À chaque itération de cette deuxième boucle, le programme récupère la valeur de la fonction *action* selon la lettre lue courant et la pile d'état, puis sur `switch(action)`, effectue les différents comportement attendu de l'automate sur la pile d'état à l'aide des fonctions préalablement déclarées. Une fois le mot accepté ou refusé, le drapeau `DONE_FLAG` est mis à `1`, permettant alors de quitter la boucle et re-itérer le processus sur une nouvelle entrée d'utilisateur.

3 Limitations

Le plus gros défaut de ce programme, à mes yeux, et l'absence de robustesse face aux fichiers passés en arguments. En effet, le programme ne fait aucune vérification de l'intégrité des fichiers .aut, et exécute le code en assumant que ce premier est juste. Le cas où le fichier n'est pas intègre résulterait probablement, tôt ou tard durant l'exécution du programme, en une erreur de segmentation, sans pouvoir indiquer clairement à l'utilisateur où se situe l'erreur dans le fichier.

Finalement, on notera également une limitation sur l'entrée utilisateur ; le buffer `user_input` est limité à 256 octets. On ne peut donc pas, à priori lire un mot de taille plus grande que 256 lettres (ou 255 lettres, en raison du comportement de `fread`) au risque de faire face à un comportement indéfini.