Rapport projet programmation impérative

Adib Habbou

9 janvier 2022

Table des matières

1. Méthode utilisée

Pour coder le programme qui exécute des automates `LR(1)` décrit dans un fichier `AUT` on commence d'abord par stocker toutes les données nécessaires des fonctions `action`, `décale`, `réduit` et `branchement` dans des matrices puis ensuite appliquer l'algorithme. Pour ce faire on implémente les piles à travers les fichiers `stack.h` et `stack.c`, et également les matrices à travers les fichiers `matrix.h` et `matrix.c` puis l'automate dans un fichier `automaton.c`. Toutes les dépendances sont gérées par un `Makefile`.

2. Implémentation des piles

J'ai choisi d'implémenter les piles à l'aide d'une liste chaînée pour ne pas avoir besoin de définir une taille fixe. J'ai donc utilisé une structure `Stack` composée d'un entier `top` qui représente le sommet de la pile et d'un pointeur `next` qui pointe vers l'élément suivant de la pile. Les fonctions et procédures que j'ai implémenter pour les piles sont les suivantes :

- une fonction qui renvoie une pile vide ;
- une fonction qui vérifie si une pile est vide ;
- une procédure qui empile un élément au sommet de la pile ;
- une fonction qui dépile l'élément au sommet de la pile et le renvoie ;
- une procédure qui affiche la pile.

3. Implémentation des matrices

Afin de stocker mes données j'ai opté pour des matrices pour pouvoir accéder à un élément grâce à deux indices vu que `action`, `décale` et `branchement` prennent deux paramètres chacun. Pour ce faire j'ai utilisé une structure composé d'un tableau à deux dimensions (tableau de tableau d'entier) ainsi que d'un entier `row` et d'un entier `column` qui représentent respectivement le nombre de lignes et de colonnes de la matrice. Les fonctions et procédures que j'ai implémenté pour les matrices sont les suivantes :

- une fonction qui créer une matrice étant donnée son nombre de lignes et de colonnes ;
- une procédure qui affiche la matrice ;
- une procédure qui libère la mémoire alloué lors de la création d'une matrice.

4. Compilation et modularité

Pour ne pas avoir à compiler manuellement tout les fichiers à chaque modification j'ai créé un `Makefile` où je décris les dépendances des fichiers. Dans mon projet, `stack.h` est inclus dans `stack.c` et `matrix.h` est inclus dans `matrix.c`, de plus `stack.h` et `matrix.h` sont inclus dans `automaton.c`. Les fichiers objets `automaton.o`, `stack.o` et `matrix.o` servent à générer l'exécutable `automaton`. J'utilise le compilateur `gcc` avec les options `-Wall` et `-Wextra`. La commande `make` permet de compiler tous les fichiers et `make clean` permet de supprimer les fichiers objets avec la commande `rm` et l'option `-f` qui permet d'ignorer les fichiers inexistants.

5. Récupération des données

Je récupère en premier le nombre d'état de l'automate en lisant la première ligne du fichier `AUT` avec un `fgets`. Le nombre d'états étant stocké dans un tableau de caractères j'utilise la fonction `atoi` (`ASCII` to integer) pour le convertir en entier.

Ensuite je récupère les valeurs de la fonction `action`, en parcourant le fichier à l'aide d'un `fread` tout en ajoutant à chaque fois la valeur stockée dans le buffer à ma matrice. La matrice `action` est donc telle que l'indice des lignes représente `s`, l'indice des colonnes représente `c` et l'élément présent à cet emplacement de la matrice est la valeur de `action(s,c)`.

Puis je récupère les valeurs de la fonction `réduit` dans une matrice à deux lignes, la première ligne correspond aux valeurs de la première composante de `réduit` et la seconde correspond aux valeurs de la deuxième composante de `réduit`. J'utilise un `fread` en ajoutant à chaque fois la valeur du buffer dans la matrice de manière à ce que `s` soit l'indice de la colonne où sont stockés les deux composantes de `réduit(s)`.

Pour récupérer les valeurs de la fonction `décale`, j'utilise encore une fois `fread` mais en lisant trois octets à la fois à l'aide d'un tableau de caractères de taille trois. Le premier élément correspond à l'indice des lignes et représente `s`, le deuxième correspond à l'indice des colonnes et représente `c` et le troisième représente la valeur de la case d'indice `s` et `c` qui correspond à `s' = décale(s,c)`.

J'opère de la même manière pour la fonction branchement que pour la fonction `décale`, je récupère ainsi une matrice où l'indice des lignes représente `s`, l'indice des colonnes `A` et la valeur de la case d'indice `s` et `A` est `s' = branchement(s,A)`.

6. Application de l'algorithme

Une fois toutes les données correctement stockées dans les matrices, je stocke la chaîne de caractères saisie par l'utilisateur dans un buffer en utilisant un `fgets`. Je crée une pile d'état que j'initialise avec la valeur 0 au sommet de la pile. Je récupère la taille de l'entrée de l'utilisateur grâce à la fonction `strlen` de la bibliothèque `string.h` puis je parcours la chaîne de caractères avec une boucle `while` en déterminant à chaque fois la valeur de la fonction `action` :

- si elle est égale à 0 je rejette, j'indique quel caractère n'est pas reconnu et où il se situe dans la chaîne saisie par l'utilisateur (quand le caractère reconnu est un espace ou un saut de ligne je gère le message d'erreur séparément par soucis de clarté pour l'utilisateur), puis je sors de la boucle ;
- si elle est égale à un 1 j'accepte puis je sors de la boucle ;
- si elle est égale à 2 je décale en empilant la valeur de `décale(s,c)` puis je passe au caractère suivant ;
- si elle est égale à 3 je réduit en dépilant n fois puis en empilant la valeur de `branchement(s,A)`.

7. Gestion des erreurs

L'utilisation de plusieurs fonctions pouvant aboutir à des erreurs (`fopen`, `fclose`, `fgets`, `fread`) m'a poussée à vérifier à chacune de leur utilisation qu'aucune erreur ne s'est produite. Si jamais une erreur a eu lieu j'affiche un message expliquant l'erreur et je sors du programme avec un `exit` en renvoyant un numéro d'erreur dont voici la liste :

- l'erreur 1 renvoie à un mauvais nombre d'arguments ;
- l'erreur 2 renvoie à un problème lors de l'ouverture du fichier ;
- l'erreur 3 renvoie à un problème lors de la fermeture du fichier ;
- l'erreur 4 renvoie à un problème lors de la lecture de la première ligne du fichier ;
- l'erreur 5 renvoie à un problème lors de la lecture du fichier ;
- l'erreur 6 renvoie à un problème lors de la lecture de la saisie de l'utilisateur.

8. Problèmes rencontrés

- Le nombre d'états que l'on souhaite récupérer n'est pas un entier dans le fichier `AUT`. Problème résolu en utilisant la fonction `atoi` qui permet de transformer une chaîne de caractères de type `char`, représentant une valeur entière, en une valeur numérique de type `int`.
- Les valeurs que l'on stockent dans un buffer après les avoir lu dans un fichier `AUT` avec un `fread` sont de type `char`. Mais les matrices que j'utilise sont des matrices d'entiers et surtout les indices des matrices doivent être des entiers. Problème résolu en mettant un `(int)` avant chaque `char` que l'on souhaite utiliser comme `int`.
- Certaines valeurs de `réduit` sont négatives dans le fichier `AUT`. Problème résolu en utilisant un buffer de type `unsigned char` qui est un type de données où la variable consomme tous les 8 bits de la mémoire et il n'y a donc pas de bit de signe (contrairement au type `signed char`). Cela signifie donc que l'intervalle va de 0 à 255 (contrairement au type `signed char` qui va de -128 à 127).
- Les valeurs qu'on souhaite extraire pour `décale` et `branchement` sont regroupées dans des séquences de trois octets et finissent par `'\255' '\255' '\255'`, donc impossible de les récupérer de la même manière que pour `action` et `réduit`. Problème résolu à l'aide d'une boucle `while` avec laquelle on parcours le fichier trois octets par trois octets avec un buffer de taille trois en vérifiant à chaque fois l'égalité avec `'\255'`.
- Pour l'automate `dyck.aut` lorsque l'entrée saisie est par exemple `'('` ou bien `'( )'` l'affichage du caractère non reconnu n'est pas réellement clair pour l'utilisateur. Problème résolu en vérifiant lorsque que l'action est de rejette si le caractère non reconnu est un saut de ligne ou un espace, en comparant l'entier renvoyé par le caractère avec 10 et avec 32 (qui représente respectivement un saut de ligne et un espace en `ASCII`).
- L'utilisateur ne peut pas saisir plusieurs entrées à la suite, il est obligé de relancer le programme chaque fois qu'il veut saisir une nouvelle entrée. La seule solution envisagée a été de mettre la boucle `while` de l'algorithme principale dans une boucle `while` ce qui n'est pas très recommandé. Problème non résolu.

9. Défauts et limites du programme

Le programme ne respecte pas totalement l'énoncé puisque les fonctions `décale` et `réduit` sont définies (ou plutôt leurs valeurs récupérées) même lorsque l'action associé à l'état et la lettre n'est pas `réduit` ou `décale`. De plus l'utilisateur ne peut pas saisir plusieurs entrées à la suite, il doit relancer le programme.

La méthode utilisée pose de réelles problèmes de mémoire, puisqu'on stockent beaucoup de données alors qu'on en utilise réellement qu'une partie. La récupération de toutes les données pose également un problème de complexité temporelle puisque l'on rallonge le temps d'exécution du programme surtout dans les cas les plus simples où il peut rejette directement.

10. Améliorations possibles

Afin de permettre à l'utilisateur de saisir plusieurs entrées à la suite, on pourrait lui demander combien d'entrées il souhaite saisir et utiliser donc une boucle `for`. On pourrait également lui demander à chaque saisie s'il souhaite en refaire une autre et utiliser alors une boucle `while` avec une condition pour sortir de la boucle.

Une méthode beaucoup plus optimisée serait de ne pas stocker toutes les valeurs des fonctions `action`, `réduit`, `décale` et `branchement` dès le début mais de plutôt d'utiliser des fonctions qui récupèrent exactement la valeur souhaitée dans le fichier `AUT` quand on en a besoin. De manière à éviter de saturer la mémoire de données inutiles et ainsi de diminuer par la même occasion le temps d'exécution du programme.

Pour implémenter ces fonctions on utiliserait la fonction `fseek` et la constante `SEEK_CUR` pour positionner le curseur à l'endroit exact dans le fichier où on souhaite lire les données. On devra donc également s'assurer qu'aucune erreur n'a lieu, un nouveau numéro pourra alors être associée aux erreurs causées par la fonction `fseek`.

11. Représentation graphique des automates

On représente chaque automate par un graphe d'états. Les sommets du graphe représentent les différents états (`Q0` étant l'état initial), les flèches noires représentent `décale`, les flèches bleues représentent `réduit`, les flèches rouges représentent `branchement` et la flèche verte représente l'acceptation du mot.

Pour tracer ces graphes on utilise le langage `DOT` qui fait partie des outils de `Graphviz`. Dans mes fichiers `DOT`, j'utilise `digraph` pour créer des graphes orientés. La syntaxe `A -> B` crée un arc du sommet `A` au sommet `B`.

L'option `color` permet de choisir la couleur des bordures d'un sommet ou d'un arc (j'utilise la couleur blanche lorsque que je veux des sommets sans bordure), l'option `fontcolor` permet de choisir la couleur du texte à l'intérieur d'un sommet ou sur un arc et l'option `label` permet de rajouter du texte sur arc.

Par soucis de simplification et de lisibilité des graphes j'ai remplacé l'alphabet en minuscule par `a-z`, l'alphabet en majuscule par `A-Z` et les chiffres de `0` à `9` par `0-9`.