Rapport du projet d'IPI

Hélène de Foras du Bourgneuf

Décembre 2021

Table des matières

1 Introduction

Le but est de faire un programme qui renvoie si un mot fait partie du langage décrit par un automate. L'automate est dans un fichier .aut, l'utilisateur doit mettre le chemin vers ce fichier après avoir lancé ./automaton. Puis il peut marquer des mots qui sont soit dans le langage (`Accepted`) soit n'y sont pas (`Rejected`). Dans ce cas, le programme indique à quel caractère est l'erreur.

2 Choix faits

Le code est séparé en trois parties (en plus du `main`) que l'on réunit dans un Makefile.

La première partie `create_automate` permet (comme son nom l'indique) de créer l'automate. Les fonctions vont remplir `my_automate` , qui sera complet à la fin de l'appel `decode_file`.

La deuxième partie `automate` permet d'utiliser l'automate avec un mot. Cela parmet de renvoyer si le mot fait partie du langage.

Et enfin la patie `utile` contient les fonctions non spécifiques au sujet : récupérer une entrée de l'utilisateur, la longueur d'une chaîne et l'égalité entre deux chaînes.

2.1 Pour décoder le fichier

Cette première partie est consacrée aux fichiers `create_automate.c` et `create_automate.h`.

Les caractères du fichier .aut sont non signés par choix, on le définit dès le début en créant le type `string`.

Pour réunir toutes les informations du fichier .aut j'ai créé une structure :

```
typedef struct automate {
    int length;
    string description;
    int n;
    list_action *actions_values;
    int first_line_break;
    int second_line_break;
    int third_line_break;
    int fourth_line_break;
    list_reduit *reduits_values;
    list_decale *decale_values;
    int first_255;
    list_branchement *branch_values;
} automate;
```

J'ai choisi de récupérer les numéros de caractères des retour à la ligne (et du premier caractère '\255'), pour pouvoir me repérer plus facilement dans le fichier. On fait cela directement dans les fonctions utilisées par `decode_file` au moment où on arrive à un nouveau retour à la ligne par exemple.

`file_to_string` utilise la fonction `fread` qui renvoie le nombre de caractère du fichier .aut, on met cette valeur dans `my_automate->length`. De plus `fread` permet de stocker les caractères du fichier .aut dans une chaine de caractère que on renvoie, puis qui est stockée dans `my_automate->description`.

La fonction `val_n` renvoie le nombre d'état que l'on met dans `my_automate->n`. Il y a au maximum 256 états, donc le nombre d'état est sur trois caractères au maximum. On stocke les caractères dans une chaine jusqu'au retour à la ligne.

J'ai utilisé des listes chainées pour représenter les actions, ainsi que les valeurs des fonctions pour réduire, décaler et faire les branchements.

Voici l'exemple de la liste des actions :

```
struct list_action{
    n_action act;
    list_action* next;
};
```

Les fonctions `make_list_action`, `make_reduit`, `make_decale` et `make_branch` fonctionnent quasiment sur le même principe. Elles sont appelées dans `decode_file` et elles renvoient toute une liste chaînée contenant les éléments correspondant du fichier .aut. On lit `my_automate->description` jusqu'au prochain retour à la ligne, en créant à chaque fois un nouveau maillon de la liste chaînée.

Dans le cas de `make_reduit`, on remplit une première fois le nombre d'états à dépiler `l_reduits->nb_etats` dans la liste chaînée, puis on revient au début de la liste chaînée pour remplir le caractère de branchement `l_reduits->sym_non_term`. Et on renvoit la liste chaînée `l_reduits`.

2.2 Pour vérifier si un mot appartient au langage

Cette deuxième partie concerne les fichiers `automate.c` et `automate.h`.

Comme demandé dans l'énoncé, j'implémente des piles d'états, pour cela on utilise des listes chaînées (encore !). On créé la pile avec `create_pile`, on empile et on dépile avec les fonctions éponymes. La structure des piles d'états est :

```
typedef struct stack_etats stack_etats;

struct stack_etats{
    etat e_tat;
    stack_etats* next;
};
```

La fonction `action` renvoie l'action associée à l'état s et au caractère c. Pour cela, on avance $128 * s + c$ fois dans la liste chaînée des actions, car c'est comme ça que ces dernières sont rangées.

La fonction `function_decale` renvoie un état en cherchant dans `my_automate->decale_values` celui qui correspond à l'état s et au caractère c. `function_branch` est basée sur le même principe :

```
etat function_branch(etat s, char sym, automate* my_automate){
    list_branchement *l_branch=my_automate->branch_values;
    list_branchement *l_first=l_branch;
    while ((l_branch->etat_s != s) || (l_branch->sym_non_term !=
    ↪  sym)){
        l_branch = l_branch->next;
    }
    etat s_branch=l_branch->etat_s2;
    l_branch=l_first;
    return s_branch;
}
```

Pour ce qui concerne `function_reduit`, on doit retourner un nombre et un caractère. J'ai fait le choix de renvoyer une `list_reduit`, contenant seulement ces deux éléments et dont le `next` est `NULL`.

La fonction principale de `automate.c` est `result`. Elle fait appel aux fonctions citées ci-dessus en prenant en argument une chaine de caractère. Elle renvoie le resultat `Accepted` ou `Rejected` (suivi d'un message indiquant à quel caractère l'erreur est le cas échéant). Elle fait appel à `free_stack`, la fonction qui libère la pile d'états grâce à `free`.

Si l'action est `Reduit`, voici la liste des actions effectuées.

```
list_reduit *l=function_reduit(pile->e_tat,my_automate);
int n_depile=l->nb_etats;
char A=l->sym_non_term;
for (int j=0;j<n_depile;j++){
    pile=depile(pile);
}
pile=empile(pile,function_branch(pile->e_tat,A,my_automate));
free_reduit(l);
```

On dépile le nombre d'état obtenus grâce à `function_reduit`, puis on fait le branchement avec le caractère non terminal. Enfin on libère la mémoire utilisée par la `list_reduits` avec la fonction `free_reduit`.

Lorsque l'utilisateur a écrit "quit" dans le flux entrant, on libère la mémoire prise par l'automate. On utilise `free_automate` qui emploie `free` en faisant des boucles `while`.

2.3 Pour lire un mot

Enfin, cette partie porte sur les fichiers `utile.c` et `utile.h`. Elle contient trois fonctions qui pourront être réutilisées tel quel dans d'autres programme.

La fonction `read_input` utilise `fgets` qui stocke la chaîne de caractère du flux entrant dans un buffer, puis elle renvoie ce buffer. Les caractères sont signés, on vérifie qu'ils sont bien ASCII (entre 0 et 127).

Dans `equal`, pour éviter d'utiliser `stdbool.h`, on utilise comme convention de test : 1 pour True et 0 pour False.

3 Problèmes techniques et solutions trouvées

Pour lire les fichiers .aut, j'utilise `file_to_string` puis `aff_string`. J'ai aussi créé des fonctions qui affichent les éléments des listes chaînées (ces fonctions ne sont pas dans les fichiers rendus). J'ai ainsi pu vérifier que les valeurs étaient cohérentes dans les listes chaînées.

Pour vérifier que la pile correspondait à ce qui était attendu j'affichais systématiquement la pile dans la fonction `result` grâce à la fonction `disp_stack` qui parcourt la pile en partant des éléments les plus récents vers les plus anciens.

J'ai débogué avec Visual Studio 2022, pour vérifier pas à pas mon code. J'ai ansi pu débusquer quelques erreurs qui m'ont bloqué pendant longtemps et les réctifier. Il y a un warning sur `fgets` que l'on peut enlever avec pragma.

Pour vérifier qu'il n'y ait pas de fuites mémoires j'ai utilisé Valgrind. Effectivement (et heureusement), il n'y en a pas.

4 Limites du programme

Le fichier .aut fait moins que `LENGTH_MAX_STRING` caractères. On peut néanmoins modifier cette valeur, pour un fichier .aut spécialement long.

Je n'ai pas testé la complexité temporelle de mon programme sur des automates très lourds. Dans le cas des exemples fournis, ça renvoie le résultat en moins d'une seconde, ce qui me convient.