

Rapport de projet IPI 2021, ENSIIE

Aurélien MAZAHÉRI-PETIT

9 janvier 2022

Table des matières

1	Présentation du projet	1
1.1	But du projet	1
1.2	Principe	2
1.3	La gestion des fonctions	2
1.4	La gestion de la pile d'états	3
2	Gestion du projet	4
2.1	Récupération des fonctions avec <code>fgetc</code> et <code>fread</code>	4
2.2	Implémentation de l'algorithme dans <code>main.c</code>	5
2.3	Bonus : implémentation des graphes d'états	6
3	Conclusion	7

Résumé

Rapport de projet de programmation impérative de 2021 ¹

1 Présentation du projet

1.1 But du projet

Le but de ce projet est d'implanter un programme en langage C qui exécute des automates à pile d'état dits LR(1) (Left to right, Rightmost derivation). Ces derniers servent à reconnaître des langages de programmation.

Les automates fournis sont 4 fichiers binaires portant l'extension `.aut` (`arith.aut`, `dyck.aut`, `word.aut`, `word _ bis.aut`). Les entiers contenus sur plusieurs lignes dans chaque fichier correspondent à une description binaire de l'automate sous formes de fonctions d'états et/ou de caractères ascii.

1. Page web : https://web4.ensiie.fr/~guillaume.burel/cours/IPI/projet_2021.html

1.2 Principe

Le programme va lire un fichier dont le nom sera passé en unique paramètre de l'exécutable. Il va ensuite lire des lignes sur l'entrée standard. Après chaque entrée lue, il indiquera si la saisie est correcte pour le langage correspondant à l'automate.

Pour implémenter l'automate schématisé en Figure 1, il faut d'abord lire et extraire les nombres entiers correspondants à la description de l'automate, à savoir le nombre d'états et les fonctions action, réduit, décale et branchement.

Ensuite il reste à créer une pile d'états afin d'appliquer ces fonctions lors de la l'application de l'algorithme sur les entrées de l'utilisateur.

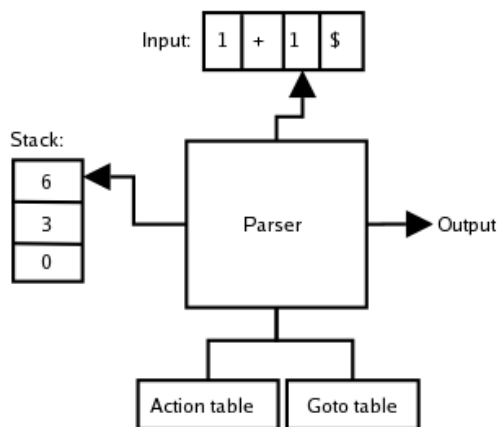


FIGURE 1 – Un automate LR1 (par exemple arith.aut)

L'automate ci-dessus part d'un état initial 0 jusqu'à un état courant 6 et applique l'algorithme à implémenter selon $action(s,c)$

1.3 La gestion des fonctions

J'ai choisi d'utiliser des **tableaux 2D** pour stocker les fonctions, dépendant souvent de 2 paramètres, notamment $action(s,c)$ comportant n fois 128 valeurs.

```
/*Cr ation d'un tableau 2D afin de stocker les donn es lues*/
matrix create_matrix(int line , int column){
    matrix mat = (matrix) malloc(line*sizeof(int*));
    int i;
    for (i=0;i<line ; i+=1){
        mat[i]=(int*) malloc(column*sizeof(int));/*chaque case 1D pointe vers un tableau 1D*/
    }
}
```

```

    return mat;
}

```

La fonction `free _matrix` permettra quant à elle d'éviter les fuites mémoires en libérant chaque tableau dans l'ordre inverse d'allocation à la toute fin des 10 saisies, nombre que j'ai choisi de limiter dans *main.c* afin de pouvoir libérer la mémoire.

```

/* Lib re la m moire utilis e */
void free_matrix(matrix mat, int line){
    int i;
    for(i = 0; i < line; i++){
        free(mat[i]);
    }
    free(mat);
}

```

1.4 La gestion de la pile d'états

Le module *stack.c* définit quant à lui la pile (stack) ainsi que des fonctions d'empilement (push), de dépilement (pop), ainsi que de récupération de dernier éléments sans suppression (last). Cette dernière m'a été utile afin de bien m'assurer que l'état courant était le sommet de la pile notamment dans la fonction `automate()` dans *main.c* pour réduit et branchement

```

#define N 256

struct stack { /*pile = le nombre d' lments et un sommet les listant*/
    int stack[N];
    int top;
};

typedef struct stack stack;

/*Cr ation d'une pile*/
stack new_s() {
    stack s;
    s.top = -1;
    return s;
}

int isEmpty(stack s) {
    return (s.top < 0);
}

```

On définit une taille fixe pour la pile de 256 éléments, sachant qu'on ne peut dépiler et empiler plus de 255 états. L'alias `typedef` permet d'alléger le code.

Pour déterminer qu'une pile est vide ainsi que pour dépiler, on se sert du fait que l'indice du top est le nombre d'éléments, ainsi on enlève 1 au top pour retirer un élément jusqu'à ce que `top=0`.

2 Gestion du projet

2.1 Récupération des fonctions avec `fgetc` et `fread`

La partie principale du projet, qui est de lire le fichier `.aut` afin d'extraire les fonctions dans les matrices a été pour moi la plus difficile. De la compréhension du format des fichiers `.aut` à la vérification des données stockées, j'ai dû à maintes reprises utiliser les utilitaires `xxd`/`hexdump` et le site <https://www.dcode.fr/fichier-donnees>

En effet, n'ayant pas l'habitude de travailler sur des fichiers binaires, j'ai passé plusieurs semaines à lire les entiers comme chaînes de caractères.

Une fois le sujet repris à tête plus reposée pendant les vacances, j'ai pu extraire assez facilement la première ligne de caractères `ascii` connaissant le motif `"a n"` avec `fscanf`; ainsi que la seconde ligne de `n` fois 128 entiers avec la fonction `fgets()` qui s'arrête au premier retour à la ligne. Pour récupérer le fichier, on ouvre avec `fopen()` en mode `rb` pour lire le binaire et ferme avec `fclose()` après traitement.

Il a toutefois fallu penser à

- remplacer le curseur pour chaque nouvelle ligne avec `fseek()`, la taille de la première ligne (6 caractères pour `arith.aut` et 5 pour les autres) + `n*128` caractères;
- bien appliquer la formule des tableaux pour le calcul des indices afin de pouvoir mettre dans la case `(s,c)` de la matrice 2D l'action correspondante lue par le tampon 1D

Pour ne pas réfléchir par rapport à la taille du tampon j'ai adopté la taille `BUFSIZ` du module `<stdio>` pour toute la suite.

Toutefois, extraire à partir de `réduit` a été plus compliqué. En effet, le premier problème a été la lecture de caractères de la taille d'un `char`. En effet utiliser `fgets()` était ici plus compliqué notamment avec la présence du retour à la ligne (10 en `ascii`) séparant les 2 composantes de `réduit`, chacune de taille `n`. Ensuite, utiliser un `char` comme tampon transformait chaque 255 en -1, ainsi j'ai choisi un `unsigned char` uniquement pour cette fonction.

Enfin, la dernière difficulté dans la récupération des données a été pour moi la compréhension des fonctions partielles décale et branchement. En effet, à mesure que chaque caractère de la saisie sera vérifiée, le parcours du graphe des états change et certains états ont besoin d'être "réduits" afin de revenir en arrière dans le graphe ou branchés à de nouveaux afin de pouvoir autoriser ou interdire certaines associations de caractères.

Ainsi ces fonctions ne sont pas définies pour tous les caractères, et la question de la taille de la matrice à choisir a posé problème. J'ai finalement choisi, comme pour la matrice des actions 2 matrices de taille n fois 128 en remplaçant les 0 par les valeurs des nouveaux états à réduire / ou être branchés aux anciens. Autre souci, j'ai dû me rendre compte que le triplet des octets 255 était converti en octal avec pour valeur 173. L'utilisation de la boucle while a été indispensable ne connaissant pas le nombre de caractère à l'avance.

2.2 Implémentation de l'algorithme dans main.c

Une fois toutes les matrices remplies, implémenter l'algorithme dans la fonction automate() a été plutôt simple. Pour la saisie des caractères de l'utilisateur j'ai récupéré chaque ligne avec fgets() sur stdin puis ai appelé automate dans une boucle for de 10 itérations.

J'ai choisi une telle boucle afin de pouvoir libérer la mémoire à la fin de l'exécution du programme (10 fois).

```
/* Application de l'automate */
char input[256];
for(int iteration=0; iteration<10; iteration++){ /* autorise 10 entrées */
    fgets(input, 256, stdin);
    automate(input, n, m_action, m_reduit, m_decale, m_branchement);
}
```

Pour le traitement des caractères, j'ai utilisé une boucle for afin d'itérer sur chaque caractère, de 0 à strlen(input), soit 2 caractères de plus que la chaîne pour permettre la saisie des caractères \n et \0 utilisés par dyck.aut et arith.aut. Ensuite, j'utilise une boucle while avec un drapeau/flag BRANCHEMENT _TEST afin de pouvoir rester sur le même caractère à l'itération suivante si l'état précédent correspondait à Réduit==3.

Pour implémenter l'état courant j'avais tout d'abord pensé à déclarer un int s=0; puis à le mettre dans la pile pour l'état initial. Malheureusement les empilements et dépilements associés aux fonctions réduit et décale n'assuraient plus que s était l'état courant, ce qui m'a empêché de faire fonctionner dyck et arith pendant une semaine. J'ai plus tard compris pourquoi word.aut et word _bis.aut fonctionnaient malgré ces erreurs car, l'état des lettres finit toujours par être accepté.

Ainsi la fonction last() permet à chaque appel de bien renvoyer le sommet de la pile et donc de lire l'état courant.

```
/* Renvoie le dernier élément */
int last(stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty, cannot pop last element");
        return 0;
    }
}
```

```

    }
    return s->stack[s->top];
}

```

2.3 Bonus : implémentation des graphes d'états

Ce bonus a été très enrichissant et m'a permis de corriger mes erreurs dans l'implémentation de l'algorithme.

Pour créer le fichier, j'ai dû ouvrir un nouveau flux avec `fopen()` en mode "w" pour écrire avec la fonction `fprintf()`. Il a fallu m'approprier la syntaxe dot du graphe, relativement simple : la liste des sommets puis la liste des arêtes.

Pour implémenter les états acceptants, j'ai remarqué que chaque accepte résultait d'un branchement qui ne donnait ni de réduit ni de décale ; malheureusement je n'ai pas réussi à supprimer les fausses arêtes acceptantes donnant des décale.

Grâce au site, j'ai pu générer des images au format png grâce à la commande : `dot -Tpng fichier.dot > fichier.png` L'idée m'est ensuite venue d'automatiser la création des images avec un petit script bash appelé grâce à la fonction `system()`, notamment pour vérifier plus rapidement mes résultats.

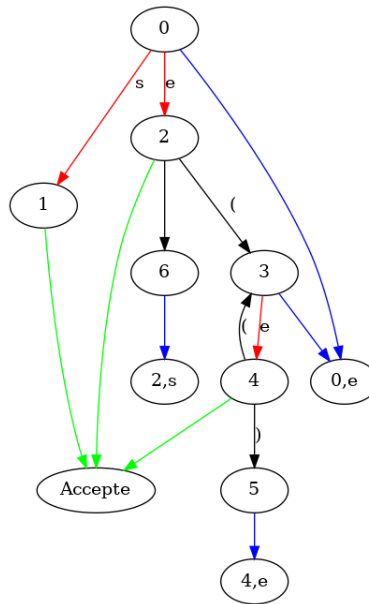
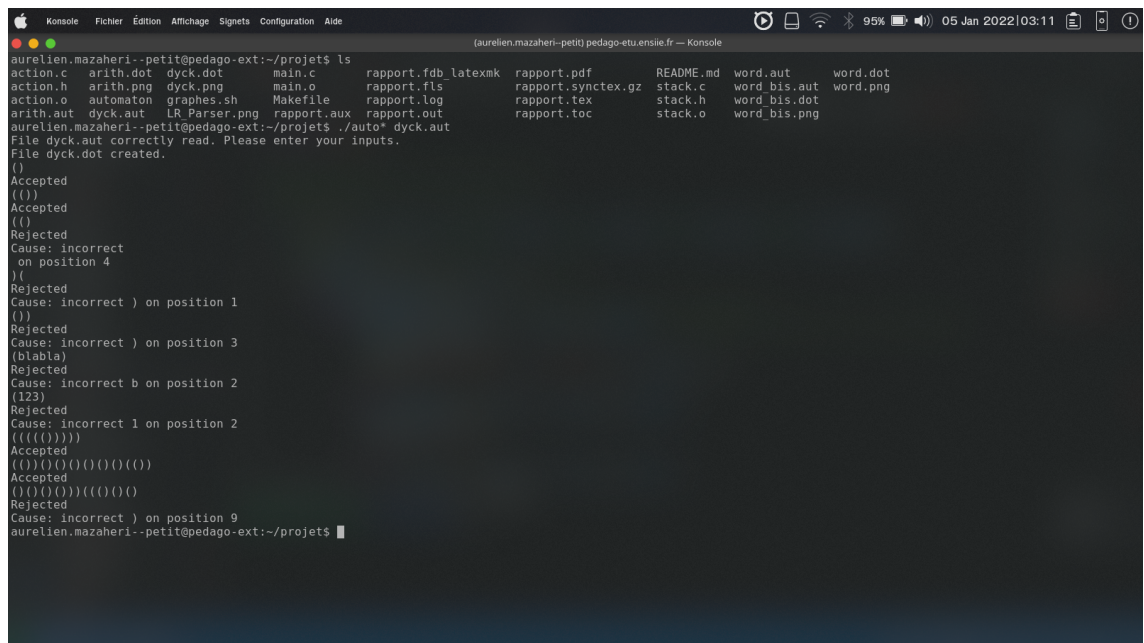


FIGURE 2 – Graphes des états de dyck.aut (seul l'état 1 est acceptant)

3 Conclusion

Des améliorations peuvent être apportées à mon projet , notamment :

- Supprimer les flèches acceptantes en trop pour dyck.aut et arith.aut, ainsi que "nettoyer" les graphes des doublons des arêtes pour word.aut et word_bis.aut ;
- une généralisation de l'algorithme pour n'importe quel fichier.aut. En effet , pour le positionnement du curseur dans la récupération des matrices, je me suis servi du nombre d'état de chaque automate comme condition afin de contourner les difficultés de lecture (notamment lire les triplets d'octets de branchement avec 2 boucles while). ;
- choisir une autre implémentation que les tableaux 2D, coûteux en mémoire et en temps d'exécution.



```
aurelien.mazaheri--petit@pedago-ext:~/projets$ ls
action.c  arith.dot  dyck.dot   main.c     rapport.fdb latexmk    rapport.pdf  README.md  word.aut   word.dot
action.h  arith.png  dyck.png  main.o     rapport.fls  rapport.pdf  rapport.synctex.gz  stack.c   word_bis.aut  word.png
action.o  automaton  graphes.sh Makefile   rapport.log  rapport.tex  rapport.tex  stack.h   word_bis.dot  word.png
arith.aut dyck.aut   LR.Parser.png  rapport.aux  rapport.out  aurelien.mazaheri--petit@pedago-ext:~/projets$ ./auto* dyck.aut
File dyck.aut correctly read. Please enter your inputs.
File dyck.dot created.
()
Accepted
{()}
Accepted
{()}
Rejected
Cause: incorrect
on position 4
){
Rejected
Cause: incorrect ) on position 1
{)}
Rejected
Cause: incorrect ) on position 3
{blabla)
Rejected
Cause: incorrect b on position 2
{123)
Rejected
Cause: incorrect l on position 2
{(((())}))
Accepted
{()}()()()()()()()
Accepted
{()}()()()()()()()
Rejected
Cause: incorrect ) on position 9
aurelien.mazaheri--petit@pedago-ext:~/projets$
```

FIGURE 3 – Exemple d'exécution pour dyck.aut