UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

ALGORITHMES POUR LA COMPILATION

Rédigé par : Mlle SATCHIVI Kokoè Yasmine Ashley

Professeur : Mr SOHIER Devan

Filière : IATIC4

Année-scolaire : 2025/2026

Table des matières

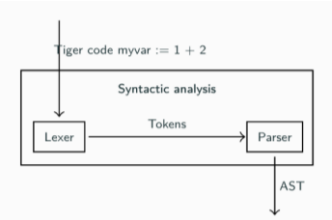
Table des figures

Table des tableaux

1 Introduction

Pour que le compilateur transforme un code source en un code cible, il passe par plusieurs phases. Après avoir reconnu les mots du langage dans l'analyse lexicale, il passe par l'analyse syntaxique ou il vérifie si la succession de mots est grammaticalement correcte avant de passer à l'analyse sémantique où il détermine le sens en prenant en compte le contexte et en levant les ambigüités de type.

La chaîne de compilation comporte plusieurs étapes dont l'analyse syntaxique est une pièce maitresse. On peut dire que l'analyse syntaxique rythme le travail du compilateur car l'analyseur syntaxique prends en entrée, comme sur le schéma ci-dessous, une succession de mots et sort l'AST (Abstract Syntax Tree ou Arbre Syntaxique Abstrait) qui est la représentation centrale utilisée par toutes les phases ultérieures du compilateur.



[1]

Figure 1 : Analyse syntaxique

Dans ce projet, nous allons nous intéresser à l' Analyse Syntaxique LR (qui est une analyse ascendante), et plus précisément à l'exécution d'un analyseur LR sur une chaine de caractère.

[1] L'image provient du cours de compilation de Mr Pablo de Oliveira Castro

2 Présentation du sujet

2.1 Contexte général

Ce projet est l'implémentation pratique des concepts théoriques étudiés. Nous implémenterons en langage C un analyseur syntaxique LR capable de prendre en entrée :

- **une chaine de caractère** à analyser ;
- **une grammaire** : elle définit les règles de production du langage à analyser ;
- **la table d'analyse LR** : elle contient les transitions d'un automate à pile.

L'analyseur simule ensuite l'exécution en effectuant des successions de décalages et de réductions. A la fin le programme doit si le mot a été accepté ou pas et doit sortir l'arbre d'analyse correspondant.

2.2 Problématique

L'implémentation d'un analyseur soulève plusieurs questions :

- Quels choix d'implémentation faire pour bien illustrer la pile et l'arbre tout en respectant les contraintes du projet ?
- Comment implémenter une pile capable de stocker simultanément les états et les symboles lus tout en prenant en compte les opérations de décalages et de dépilages multiples lors des réductions ?
- Comment construire l'arbre d'analyse au cours de l'analyse LR ?
- Comment bien afficher les actions faites, le flot à un instant t et la pile au même instant ?

Et enfin comment les différents éléments interagissent entre eux pour accepter ou refuser une chaine de caractère et afficher un arbre d'analyse ?

2.3 Objectifs

L'objectif principal du projet est d'écrire un analyseur syntaxique LR.

Spécifiquement, nous devons pouvoir :

- Charger une grammaire et une table à partir des fichiers de base ;
- Exécuter l'analyseur LR sur une chaîne de caractère ;
- Gérer la pile associée à l'analyseur LR ;
- Gérer l'arbre d'analyse de la chaîne pour la grammaire donnée ;
- Afficher si la chaîne est acceptée ou pas ;
- Afficher l'arbre d'analyse associée à l'exécution de notre analyseur.

3 Comment marche un analyseur LR ?

Nous allons illustrer son fonctionnement par un exemple.

- **Grammaire** :

Considérons la grammaire suivante :

S :a$Sb

S :

- **Table d'analyse LR** :

Tableau 1 : exemple de tableau d'analyse LR

Etats	a	b	$	S
0	d2	r2	r2	1
1			a	
2	d2	r2	r2	3
3		d4		
4		r1	r1	

- **Exécution du mot « aabb »**

Tableau 2 : exécution du mot "aabb"

Flot	Pile	Action	Commentaires
aabb	[0]	d2	• Consommer « a » dans le flot ; • Empiler « a » dans la pile ; • Aller dans l'état 2
abb	[0,a,2]	d2	
bb	[0,a,2,a,2]	r2	**Règle à réduire : « S -> ε »** • Dépiler epsilon ; Soit q l'état au sommet de la pile : • Empiler « S » • Empiler delta(q,S) Ici, q=2 et delta(q,S)=3
bb	[0,a,2,a,2,S,3]	d4	• Consommer « b » dans le flot ; • Empiler « b » dans la pile ; • Aller dans l'état 4
b	[0,a,2,a,2,S,3,b,4]	r1	Règle à réduire : « S -> aSb» • Dépiler « aSb » ; • q = 2; • Empiler "S" • Empiler delta(2,S)= 3
b	[0,a,2,S,3]	d4	
	[0,a,2,S,3,b,4]	r1	
	[0,S,1]	accepté	

Commentaire :

Lorsqu'on est dans le cas d'un décalage, on empile le caractère à lire et l'état dans lequel on arrive après avoir lu le caractère. Dans le cas d'une réduction, on dépile la partie droite de la règle sur la pile. On enfile ensuite le caractère non terminal auquel on l'a réduit puis l'état dans lequel on arrive en lisant le non terminal dans l'état au sommet de la pile.

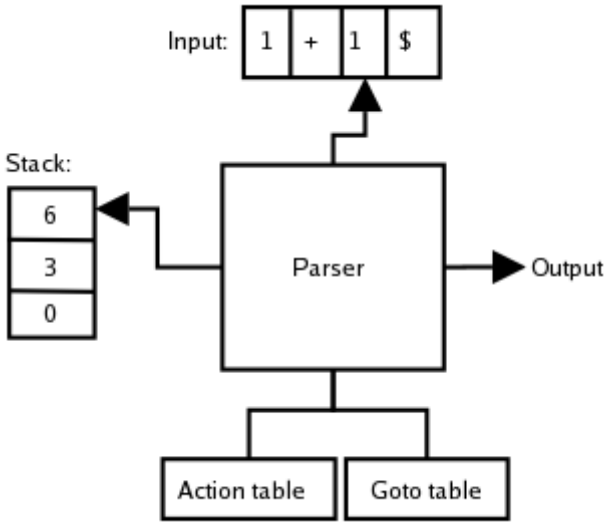


Figure 2 :. Architecture d'un analyseur syntaxique ascendant (provenant de Wikipédia)

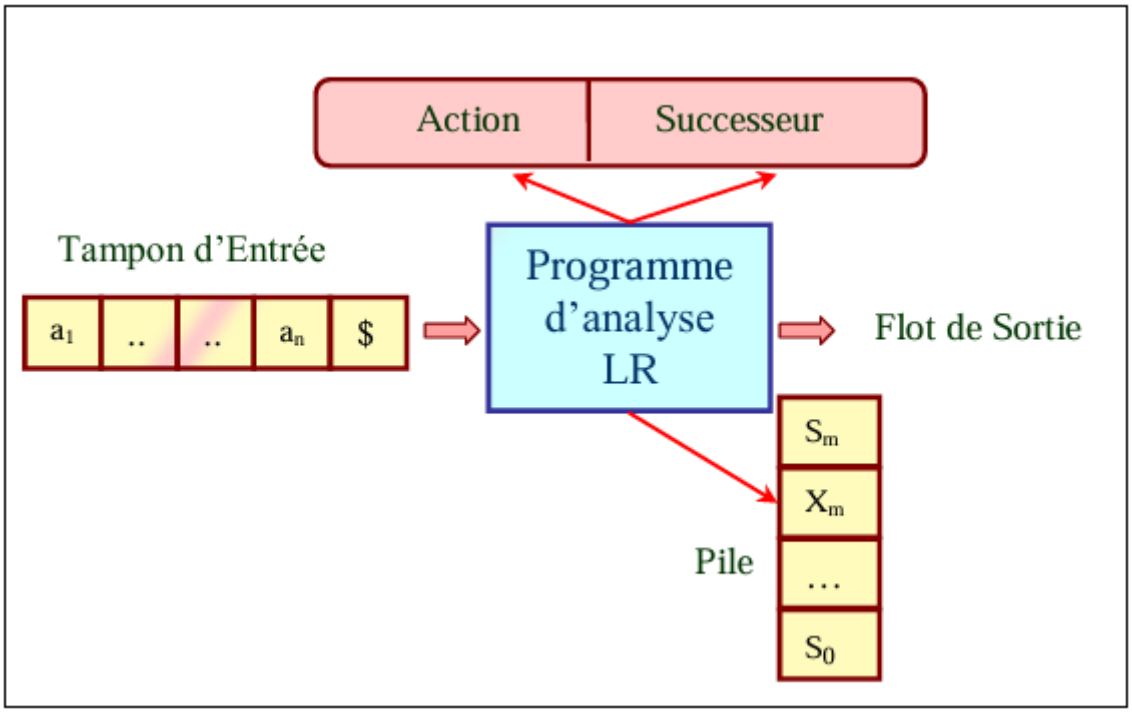


Figure 3 : modèle d'un analyseur LR (provenant du cours analyse syntaxique du professeur Souici-Meslati L.)

4 Etude de l'existant (structure de base du projet)

Pour faciliter la réalisation du projet, un ensemble de code source a été fourni. Le projet était basiquement doté, en plus des fichiers de tests, de 4 fichiers : **« LRGrammar.c »**, **« LRGrammar.h »**, **« read_file.c »**, **« read_file.h »**. Leur objectif est de permettre la lecture et la représentation des grammaires et des tables d'analyses LR.

4.1 Le fichier d'en-tête « LRGrammar.h »

Ce fichier comprend les structures de données utilisées pour représenter :

- **Struct rule** (les règles de la grammaire) composées d'un symbole non-terminal à gauche et d'une production à droite ;
- **Struct grammar** (la grammaire) qui comporte l'axiome, le nombre de règles et l'ensemble des règles de la grammaire ;
- **Struct table** : la table d'analyse LR qui comporte le nombre de lignes et un tableau de transitions ;

Le fichier comporte aussi le prototype des fonctions de manipulation de ces structures.

4.2 Le fichier source « LRGrammar.c »

- **La fonction « print_grammar() »** qui permet d'afficher une grammaire ;
- **La fonction « print_table() »** : permet d'afficher les transitions.

4.3 Le fichier d'en-tête « read_file.h »

Ce fichier comprend la structure **file_read** qui comporte une grammaire et une table. Il comprend aussi le prototype de la fonction **read_file()**.

4.4 Le fichier source « read_file.c »

La fonction **read_file()** prend en entrée le nom du fichier et retourne un objet de type file_read.

5 Pile LR

La première idée pour la gestion de la pile était un tableau. Nous avons finalement opté pour une liste chainée car :

- Un tableau ne peut pas contenir une liste hétérogène d'éléments (char pour les caractères et int pour les états) ;
- Nous ne connaissons pas le nombre d'éléments à insérer dans le tableau. Même avec un malloc, nous risquerons d'élargir le tableau au fur et à mesure.

5.1 Représentation de la pile

5.1.1 Struct NoeudPile

Chaque nœud contient :

- « w » : le carctère lu
- « q » : l'état dans lequel on se trouve ;
- « suivant » : un pointeur vers le nœud suivant.

5.1.2 Struct Pile

Elle contient une variable « tete » de type « NoeudPile » qui pointe sur le nœud correspondant au sommet de la pile.

5.2 Fonctions de la Pile

Pour bien gérer la pile des fonctions de manipulations ont été écrites.

Pile* creerPile()

Cette fonction crée une pile vide.

Int estVide(Pile* pile)

Vérifie si la vide est vide. Elle renvoie 1 si la pile est vide et 0 sinon.

Void preparerPile(Pile* pile)

Initialise la pile avec l'état 0.
Au début de l'exécution, on se trouve dans l'état 0 sans avoir rien lu. La fonction « preparerPile() » enfile donc un nœud dont l'état est 0 et le caractère lu est epsilon.

Void Empiler(Pile* p, char w, char q)

Cette fonction ajoute crée un nœud avec le caractère w lu et l'état q dans lequel on arrive après avoir lu w. elle ajoute ensuite le nœud au sommet de la pile.

Void Depiler(Pile* p)

Cette fonction retire la tête de la pile.

Char* afficherPile(Pile* p)

Elle retourne la représentation textuelle de la pile.

Le cours de la pile affiche la pile dans le sens inverse (du haut vers le bas) alors que l'on veut afficher les éléments de la pile du bas vers le haut. Pour afficher les éléments dans le bon ordre, la fonction fait un premier tour complet de la pile du compter le nombre de nœuds et déclare un tableau de nœud. On fait encore un tour supplémentaire pour recueillir les nœuds dans les cases de la table et un dernier tour pour afficher ses nœuds dans le bon ordre.

Toutes ses manipulations affectent la complexité de la fonction. (Piste d'amélioration).

Void LibererPile(Pile* p)

Elle libère la mémoire de la pile.

6 Arbre d'analyse

L'arbre d'analyse doit être construit au fur et à mesure que la pile LR est construite. contrairement à la pile, l'arbre ne stocke que les feuilles (caractères finaux). Pour ce faire, pendant l'exécution de la pile, nous allons créer une pile liée à l'arbre qui va se chargée de recueillir les informations de l'arbre d'analyse. L'arbre d'analyse est un arbre n-aire (composé de plusieurs nœuds ayant 0 ou n fils). Pour représenter la liaison entre ces nœuds, nous utilisons la représentation de Mr Jean-Michel Adam[2] via la structure NoeudArbre. Cette représentation consiste à représenter un nœud par un triplet :

- Un champ correspondant à la valeur du nœud ;
- Un champ qui pointe sur la liste des fils, plus précisément le fils ainé ;
- Un champ qui pointe sur la liste des frères, plus précisément le fils cadet.

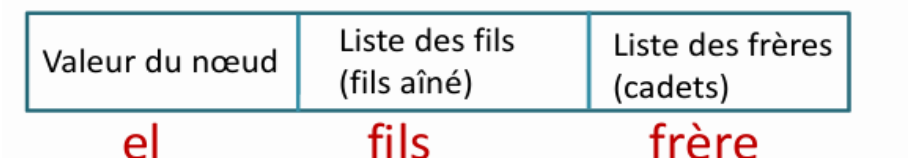


Figure 4 : champs d'un nœud (selon la représentation de Mr Jean-Michel Adam)

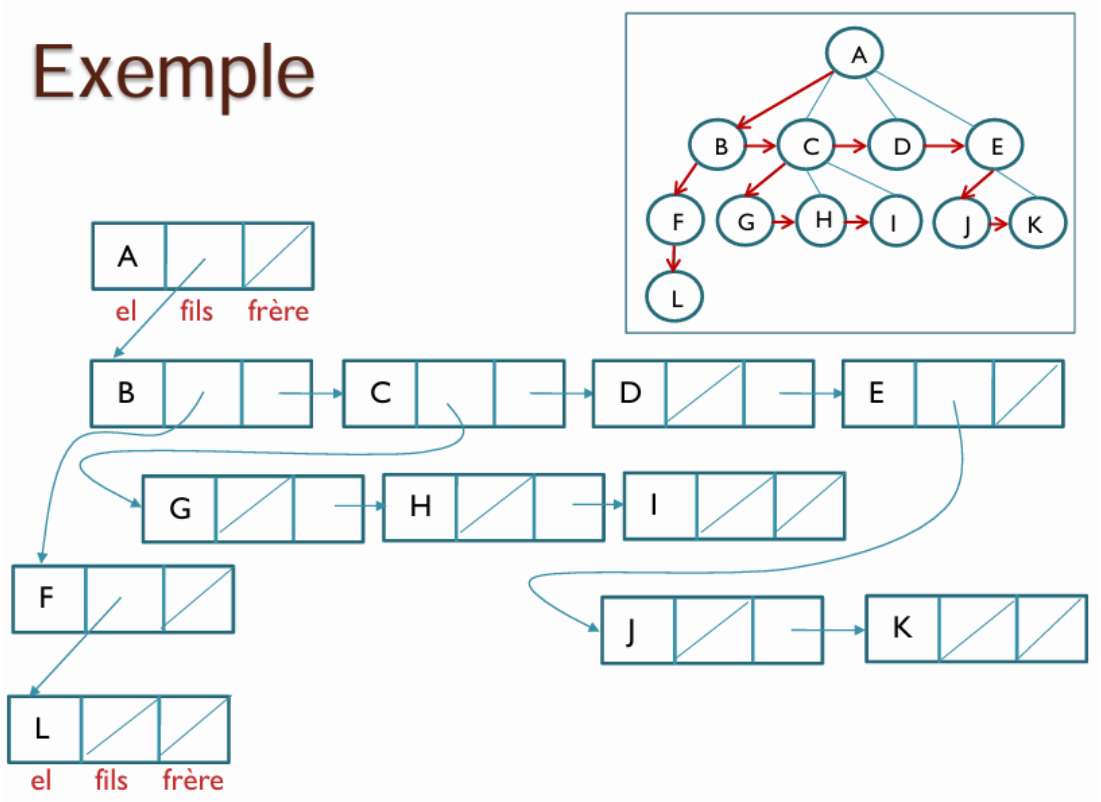
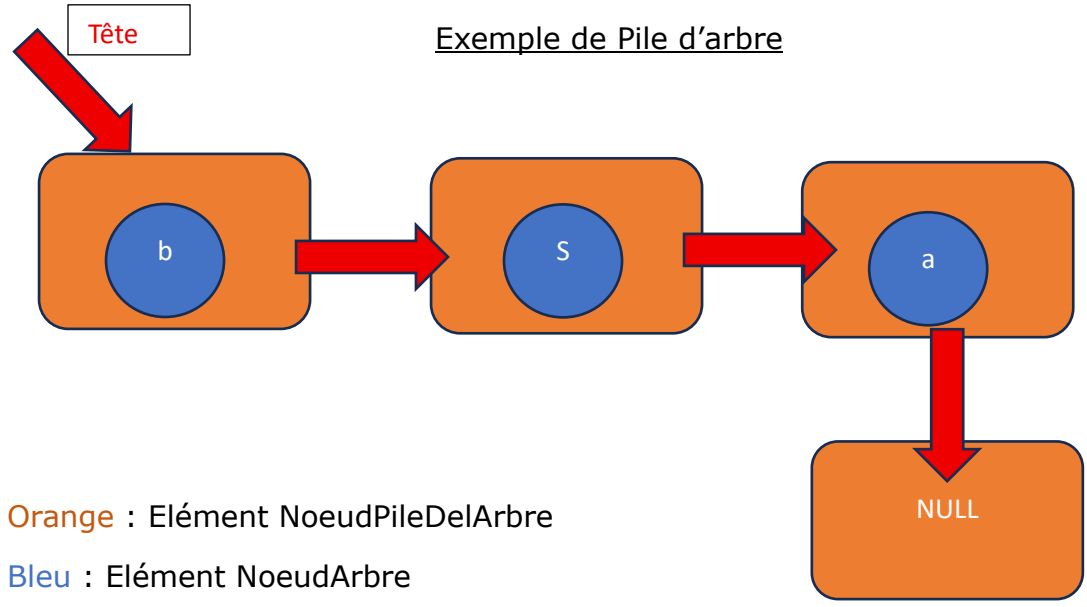


Figure 5 : Exemple visuel d'un arbre n-aire selon la représentation de Mr Jean-Michel Adam

[2] https://miashs-www.u-ga.fr/~adamj/documents/Algo24-Arbres.pdf (Jean-Michel Adam - Université Grenoble Alpes)

Cette représentation ci-dessus ne concerne que l'arbre en lui-même. Pour pouvoir lier les nœuds entre eux (ajouter des fils et frères), il faut en garder une trace et l'ordre d'apparition. Pour cela, nous avons ajouté deux autres structures :

- **NoeudPileDelArbre** qui contient 2 champs : une variable nœud de type NoeudArbre et une variable suivant pointant sur un objet de type NoeudPileDelArbre.
- **PileArbre** qui contient un champ tête pointant sur l'élément au sommet de la pile de l'arbre

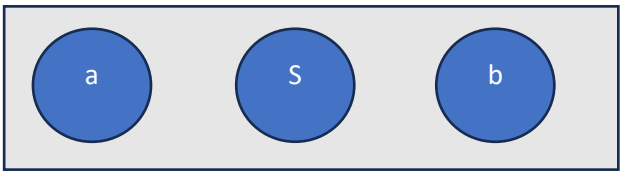


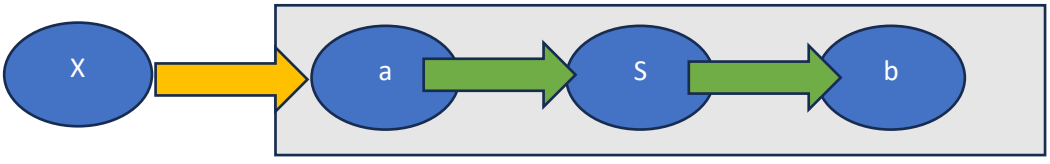
Orange : Elément NoeudPileDelArbre

Bleu : Elément NoeudArbre

: suivant

Pendant la réduction, on créera un nœud pour l'axiome. Soit X l'axiome.





: fils

: frère

6.1 Représentation de l'arbre d'analyse

6.1.1 Struct NoeudArbre

Un nœud est composé des éléments suivants :

- « valeur » : le caractère lu
- « fils » : un pointeur vers le fils ainé ;
- « frère » : un pointeur qui pointe sur le frère cadet.

6.1.2 Struct Arbre

C'est une variable « Arbre » de type « NoeudArbre* ».

Son but est juste de recueillir le Nœud racine de l'arbre pour faciliter l'affichage l'arbre.

6.1.3 Struct NoeudPileDelArbre

Son but est de pouvoir lié les éléments de l'arbre entre eux.

Un nœud est composé des éléments suivants :

- « noeud » : le nœud de l'arbre à empiler ;
- « suivant » : un pointeur vers le nœud suivant.

6.1.4 Struct PileArbre

Elle contient une variable « tete » de type « NoeudPileDelArbre » qui pointe sur le nœud correspondant au sommet de la pile de gestion de l'arbre.

6.2 Pourquoi l'utilisation des listes chainée

Une liste chainée permet d'éviter l'allocation de tableau de taille fixe qui pourrait être insuffisant ou de trop et pendant l'analyse, nous ne connaissons pas la profondeur maximale de la pile. En conclusion, elle nous permet de nous adapter automatiquement aux empilements et dépilements successifs sans redimensionnement coûteux.

6.3 Fonctions de l'arbre

NoeudArbre* creerNoeudArbre(char v);

Elle crée un Nœud arbre avec la valeur v, sans fils ni frère (elle crée une feuille).

void ajouterFils(NoeudArbre* parent, NoeudArbre* fils);

Elle ajoute le « nœud fils » comme fils du « nœud parent ».

void supprimerFils(NoeudArbre* parent);

Elle supprime le lien de parentée entre 2 nœuds.

void ajouterFrere(NoeudArbre* aine, NoeudArbre* frere);

Elle ajoute le « nœud frere » comme frere du « nœud aine ».

void supprimerFrere(NoeudArbre* aine);

Elle supprime le lien de fraternité entre 2 nœuds.

PileArbre* creerPileArbre();

Elle crée une pile vide qui va contenir les éléments de l'arbre.

int estVidePileArbre(PileArbre* pile);

Elle vérifie si la pile d'arbre est vide.

void EmpilerPileArbre(PileArbre* p, NoeudArbre* np);

Elle ajoute une feuille dans la pile de l'arbre

void DepilerPileArbre(PileArbre* p);

Elle enlève un nœud dans la pile de l'arbre.

void LibererPileArbre(PileArbre *p);

Elle libère la mémoire de la pile de gestion de l'arbre.

void afficher_Arbre(NoeudArbre *racine);

Elle permet d'afficher l'arbre. Pour le nœud racine de l'arbre, il affiche récursivement ses fils et pour chaque fils ses frères. Cette fonction est une implémentation de l'algorithme de parcours postfixé proposé par le professeur Jean-Michel Adam dans son cours sur les arbres.

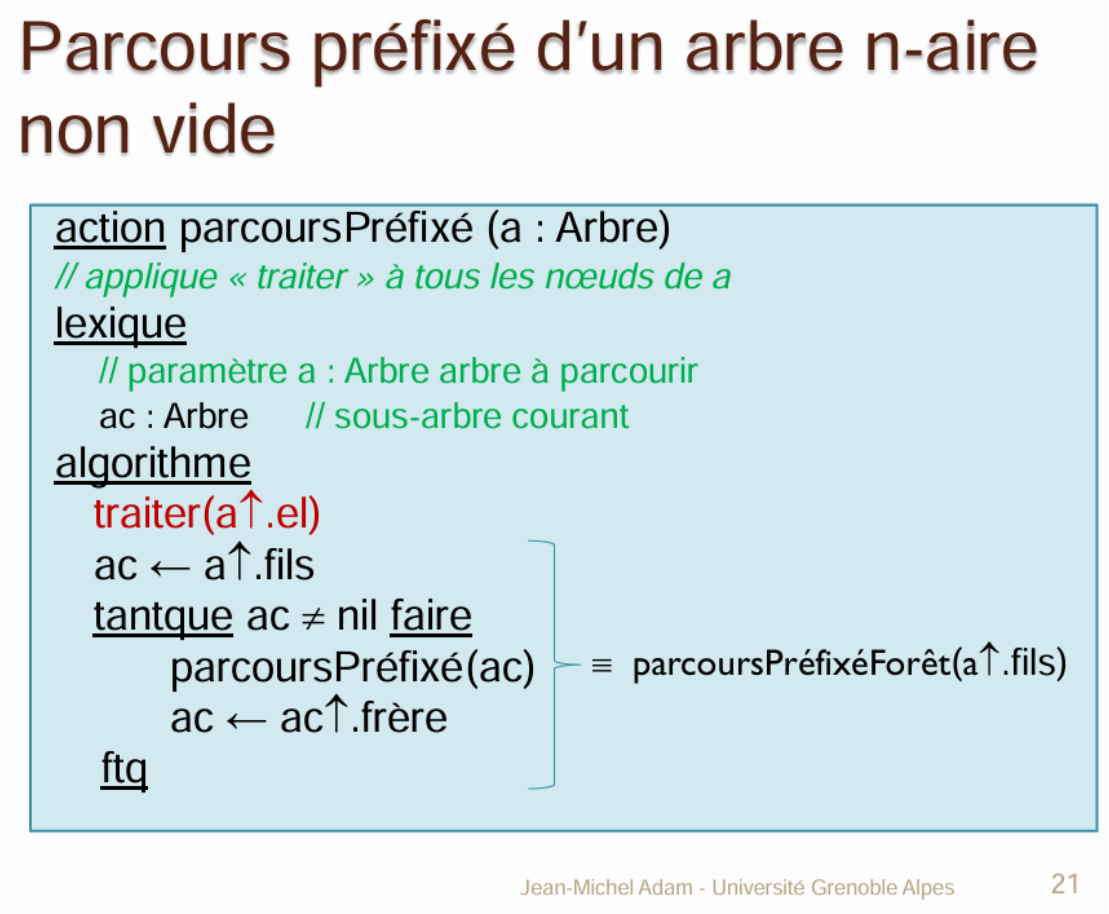


Figure 6 : Algorithme préfixé d'un arbre binaire (Pr Jean-Michel Adam)

7 Fonctions auxiliaires

Nous avons aussi certaines fonctions qui n'ont pas de liens directs avec la pile ou l'arbre dont nous avons eu besoin.

void char_to_table (const char* input, char* tab);

Elle prend la chaîne de caractère saisi et le transforme en tableau de caractères.

int nb_char_rhs(rule regle);

Elle calcule et retourne le nombre de caractère dans la partie droite de la règle (ça facilite la réduction).

char* affichage_flot(int i, int longueur_texte, char * texte_saisi);

Elle affiche les caractères du flot au fur et à mesure à partir du i-ème caractère.

void affichage_flot_pile_action(int longueur_texte, char * texte);

Elle affiche l'en-tête du tableau de pile et de flot.

void affichage_ligne_decalage(signed char action, int i, int longueur_texte, char * texte_saisi, Pile *pile0);

Elle affiche une ligne de décalage (dX, flot, pile).

void affichage_ligne_reduction (signed char action, int i, int longueur_texte, char * texte_saisi, Pile *pile0);

Elle affiche une ligne de réduction (rX, flot, pile).

8 Fonction principale (main)

Son but est de réaliser les objectifs du projet en faisant intervenir toutes les structures et fonctions écrites.

Lors de l'appel de LRanalyser sur un mot et un fichier, nous récupérons le mot saisi sous forme de liste de caractères ainsi que la grammaire et la table contenu dans le fichier.

Pour mieux réaliser l'analyse, un objet Pile est créé pour la gestion de la pile LR et un objet PileArbre est créé pour construire l'arbre au cours de l'analyse LR.

On parcourt les caractères du tableau en commençant par l'état 0 et pour chaque caractère, on a 3 choix :

- Décalage

Pour la pile LR on empile le caractère « w » lu et l'état dans lequel on arrive après avoir lu « w » sous forme de nœud.

Pour la pile d'arbre, on empile une feuille contenant le caractère lu.

- Réduction

On récupère le numéro de la règle et le nombre de caractères dans la partie droite de la règle (nombre de caractères à dépiler). Soit « N » ce nombre. On parcourt les N éléments de la partie droite de la règle (du n-ième au premier car les éléments de la pile sont stockés dans l'ordre inverse).

- Pour la pile LR :
  Si la valeur l'élément en tête de liste est compris entre 65 et 91 (caractères « A » à « Z » en code ASCII), on le rend positif car les lettres en majuscules sont représentées dans la table LR comme des entiers négatifs), sinon on laisse « w » à sa valeur d'origine. Si le caractère ne fait pas partie de la table, on arrête l'analyse. Après avoir traité la valeur de « w », on dépile le nœud en tête de la pile. On empile ensuite un nouveau nœud contenant le caractère réduit et l'état dans lequel on arrive.
  Comme aucun caractère n'est consommé dans une réduction, on décrémente l'incrément qui parcours la chaîne de caractère.

- Pour la pile de l'arbre :
  On crée un tableau de taille N de type NoeudArbre**. Son but est de recueillir les éléments qui seront dépiler dans l'arbre.
  On recueille l'élément au sommet de la pile dans les nœuds de N à 1 et on dépile la pile de l'arbre.
  On empile ensuite un nœud parent contenant le caractère non-terminal réduit. On ajoute une relation père-fils entre le nœud parent et le nœud 0. Et pour les autres nœuds, on ajoute des relations de fraternité.

- Acceptation

On arrive dans cet état quand on est à la fin du mot et que la valeur du caractère lu est « -127 ».

On affiche le mot « accept ».

- Echec

On arrive dans cet état quand il n'y a aucune action associée à la lecture d'un caractère dans un état ou si on n'arrive pas à réduire une règle.

Le contenu de la pile est afficher au fur et à mesure qu'on fait un décalage ou une réduction et l'arbre d'analyse est affichée uniquement si on est dans le cas d'une acceptation à la fin de l'analyse.

9 Exemples d'exécution

9.1 Analyse du mot « cacbcbcac » sur le fichier test4

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ cat ../test4
A:$Aa$B
A:$B
B:$Bbc
B:c
        a       b       c       $       A       B
0                       d3              1       2
1       d4                      a
2       r2      d5              r2
3       r4      r4              r4
4                       d3                      6
5                       d7
6       r1      d5              r1
7       r3      r3              r3

skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test4 "cacbcbcac"
                                Flot | Pile
      ------------------------------------------------------------
                              cacbcbcac| 0
d3                             acbcbcac| 0c3
r3                             acbcbcac| 0B2
r1                             acbcbcac| 0A1
d4                              cbcbcac| 0A1a4
d3                               bcbcac| 0A1a4c3
r3                               bcbcac| 0A1a4B6
d5                                cbcac| 0A1a4B6b5
d7                                 bcac| 0A1a4B6b5c7
r2                                 bcac| 0A1a4B6
d5                                  cac| 0A1a4B6b5
d7                                   ac| 0A1a4B6b5c7
r2                                   ac| 0A1a4B6
r0                                   ac| 0A1
d4                                    c| 0A1a4
d3                                     | 0A1a4c3
r3                                     | 0A1a4B6
r0                                     | 0A1

                               [ ACCEPT ]


A(A(A(B(c()))a()B(B(B(c())b()c())b()c()))a()B(c()))

skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

Figure 7 : analyse de "cacbcbcac" sur le fichier test4

9.2 Analyse du mot « cacbcbca » sur test4

On a enlevé une lettre pour illustrer les cas d'échec.

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test4 "cacbcbca"
                                Flot | Pile
      ---------------------------------------------------------------
                               cacbcbca| 0
d3                              acbcbca| 0c3
r3                              acbcbca| 0B2
r1                              acbcbca| 0A1
d4                               cbcbca| 0A1a4
d3                                bcbca| 0A1a4c3
r3                                bcbca| 0A1a4B6
d5                                 cbca| 0A1a4B6b5
d7                                  bca| 0A1a4B6b5c7
r2                                  bca| 0A1a4B6
d5                                   ca| 0A1a4B6b5
d7                                    a| 0A1a4B6b5c7
r2                                    a| 0A1a4B6
r0                                    a| 0A1
d4                                     | 0A1a4

                                [ REJECT]
 AUCUNE TRANSITION N'EXISTE SUR CE CARACTERE
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

Figure 8: analyse de "cacbcbca" sur le fichier test4

Comme l'analyse a échoué, on n'affiche pas d'arbre d'analyse.

9.3 Test avec le bon parenthèsage

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ cat ../test2
T:(T)T
T:
        (       )       $       T
0       d2      r2      r2      1
1                       a
2       d2      r2      r2      3
3               d4
4       d2      r2      r2      5
5               r1      r1skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test2 "(()())"
                                Flot | Pile
      ---------------------------------------------------------------
                               (()())| 0
d2                              ()())| 0(2
d2                               )())| 0(2(2
r1                               )())| 0(2(2T3
d4                                ())| 0(2(2T3)4
d2                                 ))| 0(2(2T3)4(2
r1                                 ))| 0(2(2T3)4(2T3
d4                                  )| 0(2(2T3)4(2T3)4
r1                                  )| 0(2(2T3)4(2T3)4T5
r0                                  )| 0(2(2T3)4T5
r0                                  )| 0(2T3
d4                                   | 0(2T3)4
r1                                   | 0(2T3)4T5
r0                                   | 0T1

                                [ ACCEPT ]


T(()T(()T())()T(()T())()T()))()T())

skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

Figure 9 : test du bon parenthèsage

9.4 Analyse du mot « i*i+i » sur un exemple du cours

Ne pouvant pas encore représenter un non terminal par plusieurs caractères, i représente l'identifiant(id).

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ cat ../test5
E:$E+$T
E:$T
T:$T*$F
T:$F
F:i
F:($E)
        +       *       i       (       )       $       E       T       F
0                       d4      d5                      1       2       3
1       d6                                      a
2       r2      d7                      r2      r2
3       r4      r4                      r4      r4
4       r5      r5                      r5      r5
5                       d4      d5                      8       2       3
6                       d4      d5                      9       3
7                       d4      d5                              10
8       d6                              d11
9       r1      d7                      r1      r1
10      r3      r3                      r3      r3
11      r6      r6                      r6      r6
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test5 "i*i+i"
                                Flot | Pile
      ---------------------------------------------------------------
                                i*i+i| 0
d4                               *i+i| 0i4
r4                               *i+i| 0F3
r3                               *i+i| 0T2
d7                                i+i| 0T2*7
d4                                 +i| 0T2*7i4
r4                                 +i| 0T2*7F10
r2                                 +i| 0T2
r1                                 +i| 0E1
d6                                  i| 0E1+6
d4                                   | 0E1+6i4
r4                                   | 0E1+6F3
r3                                   | 0E1+6T9
r0                                   | 0E1

                                [ ACCEPT ]


E(E(T(T(F(i()))*()F(i())))+()T(F(i())))

skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

*Figure 10 : analyse du mot « i*i+i »*

9.5 Test mot du mot vide sur le fichier test2

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test2 ""
                                Flot | Pile
      ---------------------------------------------------------------
                                     | 0
r1                                   | 0T1

                                [ ACCEPT ]


T()

skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

Figure 11 : test du mot vide sur le fichier test2

10 Affichage en latex

L'affichage de base en console présente certaines limites :

- La conservation des résultats d'analyse (ils sont effacés dès qu'on ferme la console) ;
- Le format d'affichage n'est pas très adapté pour être ajouté dans un document technique ;
- L'affichage de l'arbre est visuellement peu intuitif.

Pour combler ces lacunes, nous avons implémenté une fonctionnalité qui exporte le résultat dans un document latex.

Les structure de représentation du tableau et du dessin d'arbre ont été pris des fichiers :

- Tableau : Tableaux sous latex ;
- Arbre: Forest package for drawing linguistic trees.

Pour visualiser le contenu du fichier latex final, il faut :

- Allez sur le site https://www.overleaf.com/
- Appuyez sur new project puis sur blank project pour créer un nouveau projet vide ;
- Copiez-y le contenu du fichier « arbre_analyse.tex » puis compilez.

10.1 Architecture de la solution

Les fonctions créées ont leur signature déclarée dans le fichier « latex.h » et le contenu dans le fichier « latex.c ».

void latex_inserer_en_tete(const char *mot)

Cette fonction écrit les premières lignes d'en-tête du fichier ainsi qu'une section où est affichée le mot à analyser.

void latex_debut_declaration_tableau()

Elle écrit le début de la déclaration d'un tableau en latex.

void latex_inserer_decalage(signed char action, int i, int longueur_texte, char * texte_saisi, Pile *pile0)

Elle affiche une ligne de décalage dans le tableau de flot du fichier latex. Son fonctionnement est semblable à celui de la fonction qui affiche un décalage expliqué plus haut.

void latex_inserer_reduction(signed char action, int i, int longueur_texte, char * texte_saisi, Pile *pile0)

Elle affiche une ligne de réduction dans le tableau de flot du fichier latex. Son fonctionnement est semblable à celui de la fonction qui affiche une réduction.

void latex_fin_declaration_tableau()

Elle écrit la fin de la déclaration d'un tableau en latex.

void afficher_arbre_latex_recursif(NoeudArbre *racine, FILE *f);

Cette fonction parcourt récursivement l'arbre d'analyse et génère la syntaxe pour dessiner un arbre en latex.
Les noms terminaux ont une couleur bleue et les terminaux, une couleur rouge.

void latex_nettoyer_fichier()

Elle vide le contenu du fichier latex. Elle est utile lorsqu'on est dans l'état d'un échec. Puisqu'il n'y a pas d'arbre à afficher, on efface ce qu'on avait commencé à écrire dans le fichier.

10.2 Intégration dans le programme principal

L'exportation en latex s'effectue au fur et à mesure qu'on fait l'analyse et plus précisément lors des affichages dans la console.

En début d'analyse, on appelle les fonctions latex_inserer_en_tete() et latex_debut_declaration_tableau().

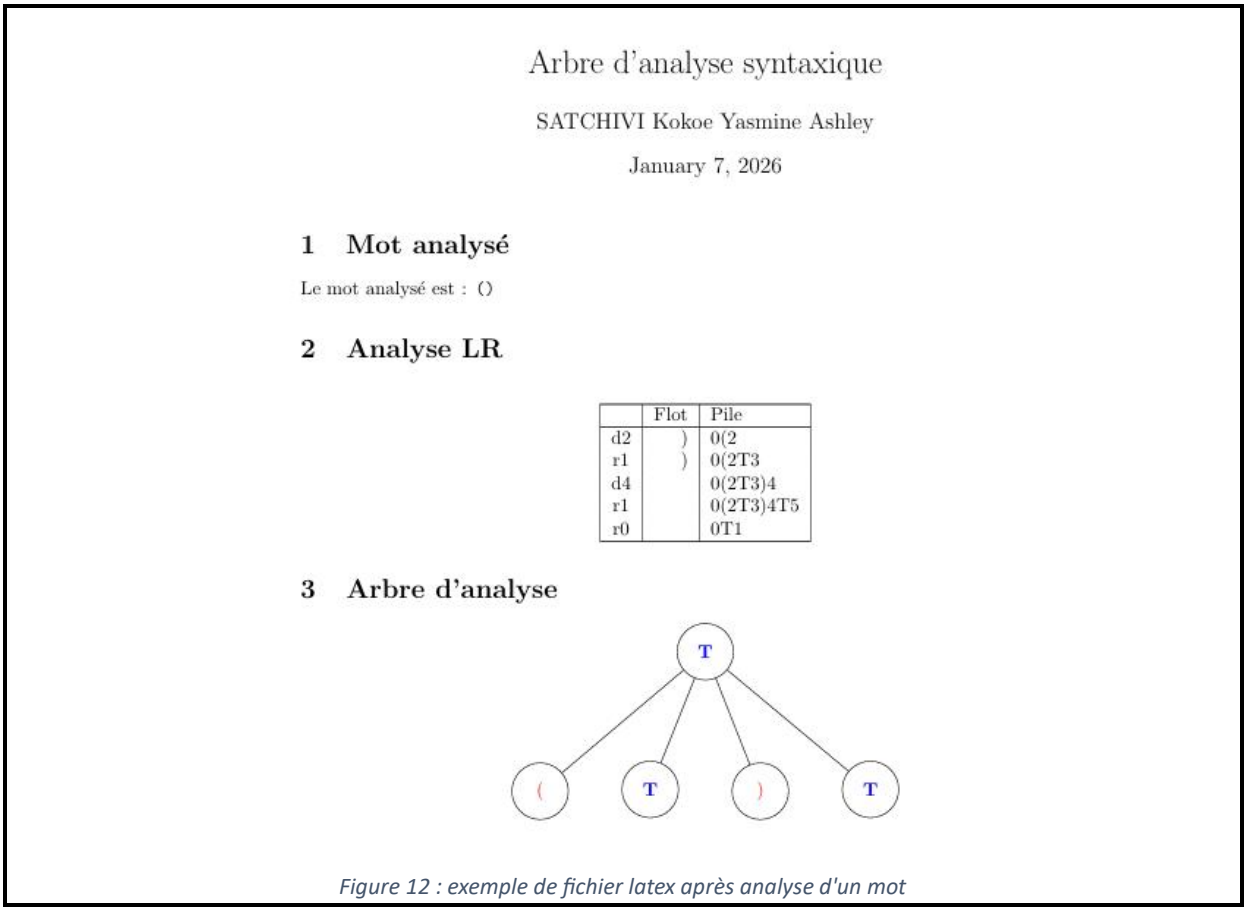
On appelle la fonction latex_inserer_decalage() lors d'un décalage et latex_inserer_reduction() lors d'une réduction.

En fin d'analyse, on appelle la fonction latex_afficher_arbre() qui insère l'arbre et fournit le fichier latex complet.

10.3 Exemple concret

Voici l'affichage de la console après l'implémentation de la l'affichage en console.

Voici, l'affichage du contenu du fichier écrit en latex.



Arbre d'analyse syntaxique

SATCHIVI Kokoe Yasmine Ashley

January 7, 2026

1 Mot analysé

Le mot analysé est : ()

2 Analyse LR

	Flot	Pile
d2	)	0(2
r1	)	0(2T3
d4		0(2T3)4
r1		0(2T3)4T5
r0		0T1

3 Arbre d'analyse

Figure 12 : exemple de fichier latex après analyse d'un mot

10.4 Limitations et perspectives d'améliorations de l'affichage en latex

- **Limitations**

Certains caractères comme « _ », « { », « } », « \ » nécessitent un échappement s'ils sont des éléments terminaux dans la grammaire. Ils constituent une limitation du code actuel et possible amélioration.

- **Améliorations possibles**
  - Gérer la compilation du fichier latex depuis le programme C ;
  - Générer le résultat sous d'autres formats (le PDF par exemple) ;

11 Problèmes rencontrés, perspectives et améliorations

11.1 Nettoyage du fichier

Lors de l'exécution des fichiers de base (test, test2, test3, test4), la fonction read_file() n'arrivait pas à lire les fichiers. Cela venait du fait que ces fichiers ont été écrits dans un autre système d'exploitation que Linux qui est le système d'exploitation que nous utilisons. Sous MAC le caractère fin de ligne est « \r » et « \r\n » sous Windows alors que sous Linux c'est « \n ».
Pour résoudre cela, nous avons utilisé la commande : **« sed -i 's/\r$//' test test test2 test3 test4 »**.
Cette commande remplace le caractère fin de ligne par le bon caractère dans nos fichiers.

Aussi, lors de la création de mon fichier test pour l'exemple du cours, l'erreur « Too many grammar rules » bien que le nombre de règles ne dépassait pas MAX_SIZE.

```
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$ ./LRanalyzer ../test6 "i*i+i"
Too many grammar rules.
skya@skya-Precision-5560:~/Documents/2025_2026/Compilation/PROJET/enonce$
```

Cela était dû au fait que le programme n'identifiait pas bien la fin de ma grammaire et le début de ma table. Pour résoudre ce problème, il faut utiliser un vrai éditeur de texte afin que les tabulations puissent bien être pris en compte par la fonction de lecture du fichier.

Pour vérifier cela, il faut saisir la commande « cat -A fichier ».

```
skya@skya-Precision-5560:~/Documents/2025_
E:$E+$T$
E:T$
T:T*F$
T:F$
F:id$
F:($E)$
    +   *   i   (   )   $   E   T   F$
0               d4  d5          1   2   3$
1   d6                      a$
2   r2  d7              r2  r2$
3   r4  r4              r4  r4$
4   r5  r5              r5  r5$
5               d4  d5          8   2   3$
6               d4  d5              9   3$
7               d4  d5                  10$
8   d6                  d11$
9   r1  d7              r1  r1$
10  r3  r3              r3  r3$
11  r6  r6              r6  r6$
skya@skya-Precision-5560:~/Documents/2025_
```

Figure 13 : exemple d'affichage de mauvais fichier test

```
E:$E+$T$
E:$T$
T:$T*$F$
T:$F$
F:i$
F:($E)$
^I+^I*^Ii^I(^I)^I$^IE^IT^IF$
0^I^I^Id4^Id5^I^I^I1^I2^I3$
1^Id6^I^I^I^I^Ia$
2^Ir2^Id7^I^I^Ir2^Ir2$
3^Ir4^Ir4^I^I^Ir4^Ir4$
4^Ir5^Ir5^I^I^Ir5^Ir5$
5^I^I^Id4^Id5^I^I^I8^I2^I3$
6^I^I^Id4^Id5^I^I^I^I9^I3$
7^I^I^Id4^Id5^I^I^I^I^I10$
8^Id6^I^I^I^Id11$
9^Ir1^Id7^I^I^Ir1^Ir1$
10^Ir3^Ir3^I^I^Ir3^Ir3$
11^Ir6^Ir6^I^I^Ir6^Ir6skya@skya-Precision-5560:~
```

Figure 14 : exemple d'affiche de bon fichier test

11.2 Affichage de la pile

Char* afficherPile(Pile* p)

La fonction de base affiche les éléments des nœuds dans le sens inverse.

Exemple : « 3c0 » au lieu de « 0c3 ».

Pour pallier ce problème, nous parcourons l'arbre 3 fois. Une fois pour compter le nombre d'éléments, une seconde fois pour allouer les éléments à afficher dans un tableau et une dernière fois qui parcoure le tableau dans le sens inverse pour donner un affichage de la pile du bas vers le sommet de la pile. Cette implémentation est en $\theta(nb\_el)$ avec nb_el le nombre d'éléments de la pile.

Pour améliorer la complexité de la fonction, on pourrait peut-être partir de la queue. Cela permettra de ne faire qu'un seul tour dans la liste chaînée.

Conclusion

Nous savons déjà que l'analyse lexicale constitue une étape centrale dans la chaîne de compilation. Ce projet nous a permis d'en voir les contours (comment se passe réellement les décalages et réductions, et comment la pile est manipulée au fur et à mesure de l'exécution) et de mettre en pratique les concepts théoriques de l'analyse syntaxique.

Au niveau de la représentation de la table, l'encodage des non-terminaux par leur code ASCII avec le bit de poids fort à 1 simplifie leurs distinctions mais le code ASCII ne peut représenter que 256 caractères ce qui nous demander comment encoder plus de caractères.

D'un point de vue technique, nous avons vu l'importance des choix des structures de données pour garantir l'efficacité du programme.

Plusieurs questions restent néanmoins ouvertes :

Que se passe-t-il lorsque la grammaire permet plusieurs arbres d'analyses pour le texte ?

Comment être plus précis dans l'indication des erreurs syntaxiques lors d'un échec ?

Peut-on illustrer les structures de données d'une autre manière de façon à améliorer la performance du programme sur des grammaires plus complexes ?

Annexes

Analyse ascendante : le principe de l'analyse ascendante est de construire un arbre de dérivation en démarrant des unités lexicales (feuilles) d'une chaîne jusqu'à arriver à la racine (qui est l'axiome de départ) en faisant des réductions successives jusqu'à retrouver l'axiome de la grammaire.

Décalage : on lit un caractère et on arrive dans un nouvel état.

Réduction : on remplace une séquence de terminaux et/ou de non terminaux reconnus par la partie droite d'une règle par un non terminal.

Analyse Syntaxique LR : c'est une méthode d'analyse ascendante qui se base sur les automates à pile. Les symboles de la chaîne sont lus de gauche à droite.

Automate : un automate est un modèle mathématique qui permet de déterminer si un mot appartient à un langage. Formellement, c'est mécanisme qui peut prendre une décision (mot accepté ou mot rejeté par exemple).

Automate à pile : Un automate à pile est une généralisation des automates. Elle a la particularité de posséder une mémoire infinie organisée en pile. Elle prend en entrée un mot et réalise une transition pour chaque lettre du mot.

Webographie et Bibliographie

- Analyse ascendante LR Lien
- Algorithme sur les arbres du professeur Jean-Michel Adam Lien
- Analyse Syntaxique ascendante : Lien
- ANALYSE SYNTAXIQUE : METHODES ASCENDANTES (professeur Souici-Meslati L) : Lien
- Wikipédia : Lien
- Tableau en latex : Lien
- Arbres en latex : Lien