COMPILATION

COURS, EXERCICES ET PROJETS DE PROGRAMMATION

Professeur Faissal OUARDI

Courriel : f.ouardi@um5r.ac.ma

Année universitaire 2023-2024

Contents

Liste des figures

Liste des tables

Liste des algorithmes

Introduction

À l'attention des lecteurs

L'écriture des compilateurs est à la fois un sujet fascinant, fastidieux et complexe. En effet, le développement des compilateurs est classée dans le top topics les plus inextricables en informatique suivant immédiatement celui de l'écriture des systèmes d'exploitation. De ce fait, l'étudiant doit être pourvu d'un ensemble complet et varié d'outils qui va du pur fondamental (théorie des langages, théories des graphes, des flots de données, optimisation, etc.), au plus palpable (structures de données, processeurs, jeux d'instructions, caches, etc.). Ainsi pour maîtriser cette complexité élevée, ce cours est de coutume scindé en deux grandes parties. La première concerne le travail antérieur accompli par un compilateur dit aussi la partie analyse ou le front-end, qui comprend les phases d'analyse lexicale, d'analyse syntaxique et l'analyse sémantique. La deuxième partie, quant à elle s'occupe de la seconde moitie des tâches d'un compilateur, baptisée la partie synthèse ou le back-end, qui prend en charge : l'optimisation indépendante à la machine, la génération de code et en fin son optimisation au égard de la machine cible.

Dans cette optique, ce rapport aborde les différentes phases de la compilation. Il est destiné aux étudiants de la troisième année de la Licence Fondamentale en Sciences Mathématiques et Informatique.

La documentation sur le développment des compilateurs est abondante. Néanmoins, ce manuscrit puise de quelques références de base tel que le fameux dragon book [4] de l'équipe du professeur Jeff Ullman, des classiques livres de J. P. Tremblay [32], de M.L Scott [29] et de Cooper & Torcson [10]. J'ai aussi beaucoup inspiré et exploité le cours en ligne du professeur Alex Aiken de l'université de Stanford [6].

Le lecteur trouvera également dans ce document une liste intéressante d'exercices et de projets de programmation par chapitre dont l'accomplissement et l'élaboration confirme le niveau de maîtrise des points abordés. Dans la liste bibliographiques, j'ai veuillé à citer les "seminal" papiers sur les travaux fondateurs de l'analyse des langages de programmation, et les liens vers les outils les plus connus dans ce domaine tel que Flex, Bison, CUP, JavaCC, etc.

L'imperfection est humaine et ce travail ne fait pas l'exception. Par conséquent, je souhaiterais aimablement inviter les lecteurs de ce document de me signaler toute erreur quelque soit sa nature (orthographe, typographie, méthodologie-style, fond, etc.)

Contexte d'apparition

La révolution industrielle de la moitie du 20^{e} siècle à fait naître les premiers instruments et calculateurs triviaux. Ces machines sont caractérisées par leurs coût et taille souvent prohibitifs mais aussi par des capacités de mémorisation et de calcul simples. Un autre aspect qui compliqua leur exploitation est leur mode d'exploitation qui reste soit manuel ou repose sur des langages machines (suite de bits) très difficile à mener et à maintenir. La figure suivante montre un code machine allégé et codé en héxadécimal pour un Pentium x86.

```
55 89 e5 53 83 ec 04 83 e4 f0 e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00 00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00 8b 5d fc c9 c3 29 d8 eb eb 90
```

Figure 1: Code héxadécimal du programme d'Eclid calculant le PGCD de deux entiers [29]

Les gens ont rapidement constaté que les programmes en codes machines sont difficiles à lire, écrire et maintenir car ils sont sujet à l'erreur. Le besoin de l'introduction d'une notation plus gérable est devenu une nécessité. Ainsi les langages d'assemblage sont créés, qui permettent de surpasser le code machine vers des codes mnémoniques plus simples.

```
/* Code en assembleur de X = (Y+4)*3;*/
  MOV EAX, Y
  ADD EAX, 4
  MOV EBX, 3
  IMUL EBX
  MOV X, EAX
```

Figure 2: Portion de code en C^{++} et l'équivalent en assembleur

Travailler en assembleur garde toujours le programmeur dépendant des détails de bas niveau de la machine cible (jeux d'instructions, de registres, etc.), où le

programme utilise un nombre important d'instructions primitives. Les progrès constants ont permis l'introduction d'instructions familiers à l'utilisateur par l'invention des premiers langages de programmation de haut niveau comme : Fortran (destiné aux calculs scientifiques), Cobol (dédié à la gestion) et Lisp (pour le calcul symbolique). Plus tard, des langages indépendants de la structure de la machine offrant des abstractions de haut niveau fut introduits tel que : C^{++}, Java et Python.

```
/* Aire d'un triangle abc en Assembleur*/
LOAD R1 a ; ADD R1 b ; ADD R1 c ; DIV R1 #2 ; LOAD R2 R1 ;
LOAD R3 R1 ; SUB R3 a ; MULT R2 R3 ;
LOAD R3 R1 ; SUB R3 b ; MULT R2 R3 ;
LOAD R3 R1 ; SUB R3 c ; MULT R2 R3 ;
LOAD R0 R2 ; CALL sqrt

/* Aire d'un triangle abc en Java */

public double aire_triangle (double a, double b, double c)
{
double s = (a+b+c)/2 ;
return Math.sqrt(s*(s-a)*(s-b)*(s-c)) ;
}
```

Figure 3: Aire d'un triangle abc en Assembleur et Java [11]

Si l'introduction de langages de haut niveau a simplifié la tâche au programmeur, les machines cibles demeurent uniquement communicables via leur propres langages (machines). Cette situation a donc parallèlement augmenté le fossé entre les deux niveaux de langages (évolués et machine), ce qui a compliqué la fonction des programmes chargés d'assurer le passage entre les deux. Ces derniers sont appelés compilateurs.

Structure générale d'un compilateur

Avant de décrire la structure globale d'un compilateur, différencions d'abord des programmes proches de ceux-ci : Interpréteur, Traducteur, Éditeur et Environnement de développement intégré [4, 11, 29].

Définition 1 (Éditeur)**.** *Une application qui permet la saisie, la modification et la sauvegarde de textes de programmes. Ex. : Edit, Bloc Notes, vi, emacs...*

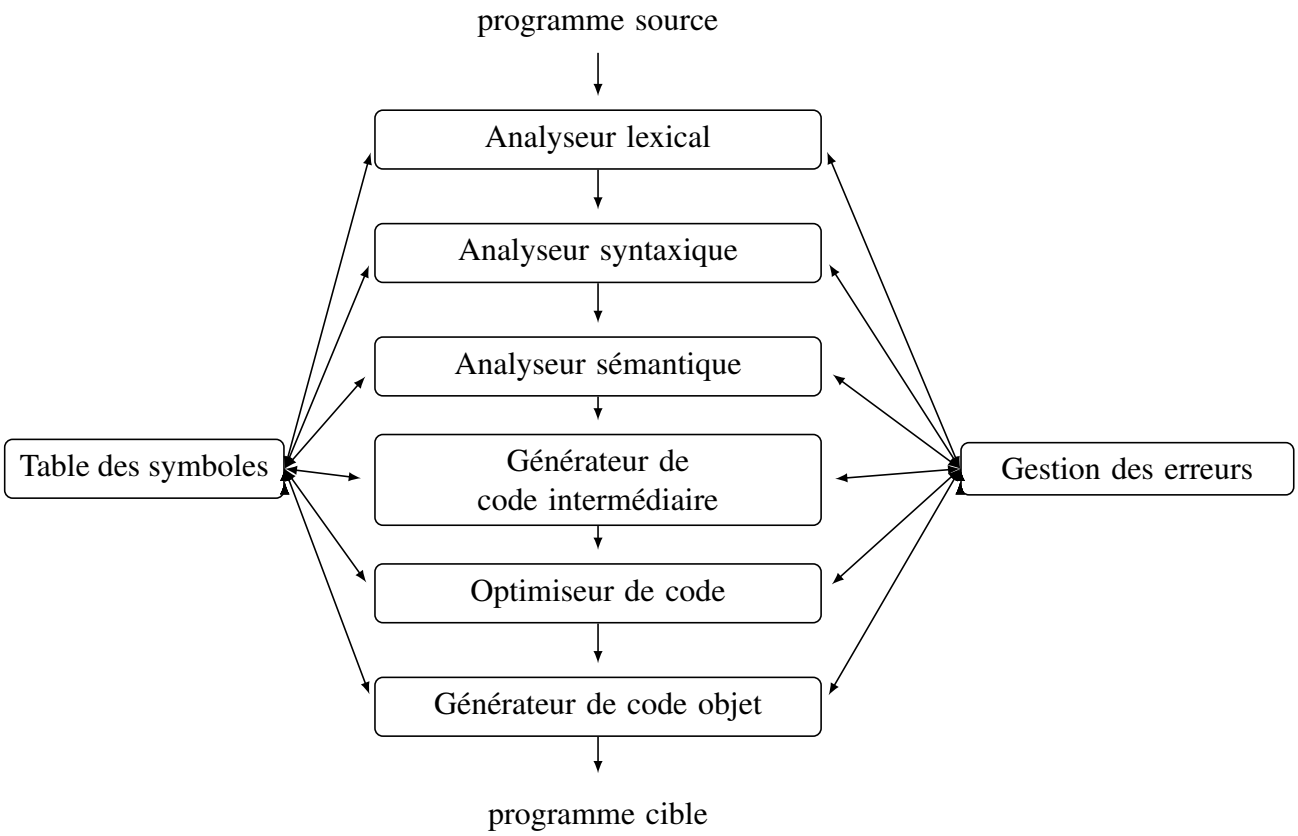


Figure 4: Architecture générale d'un compilateur

Définition 2 (Interpréteur)**.** *Traduit et exécute instruction par instruction un programme écrit en un langage évolué : Python, PHP, shell d'Unix, Perl...*

Définition 3 (Traducteur)**.** *Traduit un programme écrit en un langage évolué L1 dit langage source en un programme équivalent en un autre langage évolué L2 dit langage cible.*

Définition 4 (Compilateur)**.** *Traduit un programme écrit en langage évolué en un code exécutable équivalent en code machine. Ex : C, Pascal...*

Définition 5 (Environnement de développement intégré (EDI))**.** *Un programme complet qui intègre éditeur, compilateur avec d'autres outils de mise au point. Ex. : JBuilder, Delphi, Eclipse, IDLE, IntelliJ, etc.*

La complexité du travail d'un compilateur a suscité sa décomposition en plusieurs phases avec des interfaces bien définies comme le montre la Figure 4.

Chaque phase est a son tour décortiquée en tâches plus spécifiques. Nous présentons brièvement dans cette section un aperçu général des différentes phases. Considérons le code de calcul du PGCD de deux entiers montré dans la Figure 5, et expliquons les différents traitements qu'il (ou ses transformés) va subir durant ce long processus.

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

Figure 5: Programme source du PGCD en C [29]

Analyse lexicale

La mission de l'analyse lexicale est de regrouper les caractères du texte source du programme en unités ayant une signification du point de vue spécification lexicale du langage. Ainsi la succession de i,n,t, et espace par exemple est perçue comme une unité lexicale mots-clé "int". De cette manière, l'analyse lexicale transforme le code source d'une suite de caractères en une suite d'unités lexicales qui constitue l'entrée de la phase suivante. Il est du ressort de cette phase aussi l'insertion et le codage des unités lexicales reconnues dans une structure centrale appelée la table de symboles, la suppression des espaces, blancs et commentaires et en fin de signaler les erreurs d'ordre lexical détectées (la mauvaise formation des unités lexicales).

```
int   main int ( ) ...while...putint...}
```

Figure 6: Quelques unités lexicales du code source PGCD

Analyse syntaxique

À la réception des unités lexicales de la phase précédente, le rôle de l'analyse syntaxique est de vérifier la bonne disposition des unités reçues pour former des instructions valides du langage c-à-d conformes à sa définition syntaxique donnée

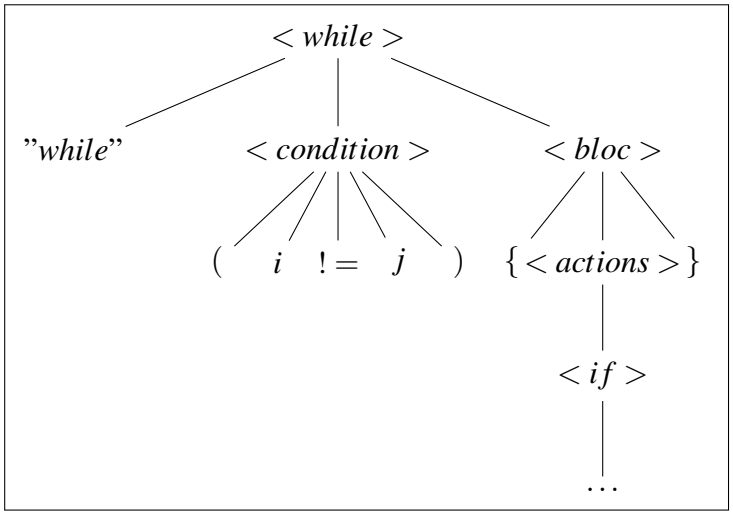


Figure 7: Arbre syntaxique d'une portion du code PGCD

par une grammaire (collection de règles du genre $A \rightarrow \alpha$). Pour justifier la validité syntaxique du code, cette phase crée un arbre dit syntaxique dont les nœuds internes sont les opérations et les fils associés les arguments. En cas d'incohérence des erreurs d'ordre syntaxiques seront émises. La figure ci-dessous montre l'arbre de l'instruction **while** du code précédent.

Analyse sémantique

Ici l'objectif est de vérifier des aspects ayant trait à la signification du code. Cette phase utilise l'arbre syntaxique fournit par la précédente et la table de symboles pour l'annoter avec les attributs consignées pour effectuer des vérifications liées au contexte comme : les déclarations, la compatibilité des types, le respect des portés, du nombre/types des arguments dans les fonctions, etc. Souvent ces aspects sont complexes à traiter avec des grammaires hors contextes comme dans la phase précédente. On fait recours à des techniques (grammaires attribuées) permettant d'opérer des actions sémantiques par des routines à invoquer dans des endroits/moments appropriés. L'analyse sémantique transforme l'arbre syntaxique en arbre abrégé et annoté ou en une forme intermédiaire pour une machine abstraite et la transmet à la suite du processus de compilation. Les erreurs d'ordre sémantiques sont signalées dans cette phase moyennant le module de gestion des erreurs.

Optimisation

Dans la première opération d'optimisation (indépendante de la machine), le code intermédiaire subi plusieurs transformations, qui préservent son logique, permettant d'améliorer son efficacité. Le gain obtenu concerne divers aspects tel que : le temps d'exécution du programme, sa taille, voire même les communications et l'énergie qu'il dispense. Après la génération du code machine, ce dernier fera l'objet d'une deuxième phase d'optimisation mais cette fois dépendant de l'architecture cible.

Génération de code

Cette phase produit le code objet en prenant en compte l'architecture et le jeux d'instruction du processeur cible. C'est une traduction du code intermédiaire optimisé vers une séquence d'instructions machine qui réalisent le même travail. Des questions à résoudre sont par exemple : attribuer des adresses (ou un mécanisme d'adressage) pour les objets manipulés, allouer judicieusement les registres de la machine souvent en nombre limité, bien sélectionner/ordonner les instructions, etc.

CHAPTER 1

Rappel sur les langages formels

1.1 Fondements

La théorie des langages formels se fonde sur plusieurs autres théories mathématiques. On fait usage ici notamment de la théorie des ensembles, de la logique mathématique et de quelques aspects de l'algèbre et ses structures élémentaires. Des notions de la théorie des graphes sont souvent exploitées également dans ce cours. Le lecteur avide est invité à consulter les diapositifs du cours dispensé en présentiel en deuxième année licence informatique ou d'exploiter les références bibliographiques sur chaque volet sus-mentionné. Le livre [20] est un bon survol de ces concepts.

Nous rappelons succinctement dans ce chapitre les base afférentes directement aux langages formels. Plus particulièrement, nous nous focalisons sur les langages réguliers. Pour plus d'exemples, de démonstrations et d'exercices veuillez consulter le livre du groupe du professeur Jeff Ullman [16]

1.2 Alphabet, mots et langages

On définit un **alphabet** comme étant un ensemble fini non vide de symboles (lettres ou caractères) qu'on note en utilisant souvent une lettre grecque majuscule comme Σ.

Sur un alphabet Σ on appelle un **mot** (chaîne ou string) w toute suite finie et ordonnée (on dit aussi séquence) de symboles de Σ. Le mot **miroir** ou transposé d'un mot w est noté w^R ou $\tilde{w}$ est obtenu en inversant tous les symboles de w.

$|w|$ dénote la longueur d'un mot w. il représente le nombre de symboles qui constituent le mot w. Le mot ayant une longueur nulle est dit le mot vide qu'on note ε. La longueur indicée par un symbole est le nombre d'occurrences de ce dernier dans le mot considéré. Par exemple $|2020|_0 = 2$ car 0 apparaît deux fois dans la

chaîne 2020.

w, x, y étant trois mots sur un alphabet Σ. On dit que :

- u est **préfixe** de w s'il existe $x \mid w = ux$
- u est **suffixe** de w s'il existe $x \mid w = xu$
- u est **facteur** de w s'il existe x et $y \mid w = xuy$

Un préfixe, (resp. suffixe ou facteur) est dit **propre** lorsque il est différent du mot vide ε et du mot w lui même

Pour un alphabet Σ l'ensemble Σ^k inclut tous les mots sur Σ de longueur k. À tire d'exemple sur l'alphabet $\Sigma = \{a,b\}$, $\Sigma^2 = \{aa,ab,ba,bb\}$. Par convention, pour tout alphabet Σ, nous admettons que $\Sigma^0 = \{\varepsilon\}$.

L'ensemble de tous les mots sur un alphabet Σ sera noté Σ^*. Ainsi, nous avons :

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots = \bigcup_{k \geq 0} \Sigma^k$$

Sur Σ^* est défini une opération dite de **concaténation** entre des mots. si u et v sont deux mots de Σ la concaténation de u et de v donne le mot uv formé des symboles de u suivis de ceux de v. Cette opération est associative et admet ε comme élément neutre. Notez bien que cette opération n'est pas en général commutative.

Un **langage** sur un alphabet Σ est tout ensemble de mots sur cet alphabet. Ceci dit, un langage L sur Σ est un sous ensemble de Σ^*. Voici quelques remarques à bien avaler sur les langages :

1. Un langage étant un ensemble, il peut être fini ou infini
2. Il n'est pas exigé que les mots d'un langage sur un alphabet inclut tous les symboles de cet alphabet
3. Tout langage d'un alphabet est aussi un langage sur son super-alphabet
4. $\emptyset$ le langage vide, et le langage formé uniquement du mot vide ε sont deux langages sur tout alphabet
5. Σ^* est un langage sur tout alphabet Σ

Les langages étant des ensembles, les opérations ensemblistes usuelles (union, intersection, complément, etc.) y sont définies de la même manière Nous insistons ici en particulier sur les opérations de concaténation et d'itération.

La **concaténation** de deux langages M et N est le langage formé des mots obtenus par concaténation des mots des deux langages en respectant l'ordre de concaténation. Cette opération est associative mais, comme pour les mots, n'est pas commutative. Elle admet le langage vide comme absorbant et le langage formé seulement du mot vide comme élément neutre.

L'itération ou l'étoile d'un langage offre un moyen de formé des mots du même langage par plusieurs succession de concaténation. En général on note :

$$\begin{cases} L^0 &= \{\varepsilon\} \\ L^n &= L.L^{n-1} \end{cases}$$

Il existe plusieurs propriétés et résultats sur les opérations sur les langage, consultez la bibliographie pour plus de détail. Nous donnons ci-après des propriétés remarquables de l'opération d'itération.

$$\begin{cases} L^* &= \bigcup_{i\geq 0} L^i \\ L^* &= L^{**} \\ L^* &= L^*L^* \\ L(ML)^* &= (LM)^*L \\ (L\cup M)^* &= (M^*L)^*M^* \\ \emptyset^* &= \{\varepsilon\} \\ \{\varepsilon\}^* &= \{\varepsilon\} \\ L^+ &= LL^* = L^*L \end{cases}$$

1.3 Grammaires

Les grammaires sont un outil permettant de générer des mots en offrant un système de réécriture sous la forme de règles dites de production. Elles permettent ainsi de définir des langages. Une grammaire est spécifiée comme un quadruplet
$G =< T,N,S,P >$ constituée d'un ensemble T de symboles **terminaux** entrant dans la composition des mots appelé aussi alphabet, d'un ensemble N de symboles **non**

terminaux également désignés de variables servant comme intermédiaires dans le processus de génération, d'un symbole spécial S élément de N appelé l'**axiome** qui joue le rôle d'initiateur et en fin d'un ensemble P de **règles de réécriture**. Il est à noter que les ensembles T, N et P sont non vides et que les deux premiers sont évidemment disjoints.

Les éléments de P sont des couples (α, β) tel que $\alpha \in (T \cup N)^* N (T \cup N)^*$ et $\beta \in (T \cup N)^*$ qu'on note en insérant une flèche entre les deux : $\alpha \to \beta$ pour signifier α peut être remplacé par β. La première partie est aussi appelé membre gauche de production et alors que la deuxième est qualifiée de membre droit de production.

On appelle une **dérivation** l'opération de remplacement d'un membre gauche par un de ses membres droits selon les règles de production de la grammaire.

$$\text{Si } A \to \gamma \text{ est une règle de P : alors } \alpha A \beta \Rightarrow \alpha \gamma \beta \text{ est une dérivation.}$$

Si un mot g dérive de f en une seule étape ($f \Rightarrow g$) on dit que g dérive directement de f. Dans le cas contraire (dérive en plusieurs étapes) on dit qu'il dérive indirectement de f et on écrit ($f \overset{+}{\Rightarrow} g$). Par ailleurs, un mot f de T^* est généré par une grammaire G s'il peut être dérivé à partir son axiome S. autrement dit :

$$G \text{ génère } f \text{ ssi } S \overset{+}{\Rightarrow} f$$

Ceci nous conduit au concept du langage généré par une grammaire qui est donc l'ensemble de mots générés par cette grammaire. On écrit:

$$L(G) = \{w \in T^* \mid S \overset{+}{\Rightarrow} w\}$$

Pour illustrer le processus de production de mots, les étapes sont schématisées par un arbre dit **arbre de dérivation**. Cet arbre possède l'axiome comme une racine et les éléments de N comme nœuds internes. Les feuilles quand à elles sont représentées par les terminaux de T. Les branches de l'arbre de dérivation schématisent les membres droits de production. à la fin d'un processus de génération la concaténation des symboles des feuilles de gauche à droite représente le mot produit associé à cette dérivation.

Une grammaire est **ambiguë** s'il existe un mot dans son langage ayant deux arbres de dérivation différents. Dans la cas contraire, elle qualifiée de **non ambiguë**.

Durant son étude sur les langage et leur structures syntaxiques **Noam Chomsky** aboutira à une classification des langages et des grammaires associées selon le degré de complexité connue sous le terme **hiérarchie de Chomsky** [9]. Cette hiérarchie inclut quatre classes de langages, les voici :

1. **Langages réguliers**, rationnels ou linéaire à droite ou encore de type 3. Ce sont les grammaires où toute les productions sont de la forme :
   $A \to \alpha B$ ou $A \to \alpha$ avec : A et B des élément de N et α une chaîne terminale de T^*

2. **Langages à contexte libre**, hors-contextuels ou algébriques ou encore de type 2. Si toutes les règles sont de la forme : $A \to \alpha$ avec $A \in N$ et $\alpha \in (T \cup N)^*$

3. **Langages à contexte lié** ou contextuels si toutes les règles de P la forme : $\alpha A \beta \to \alpha w \beta$ où : $A \in N$, $\alpha, \beta \in (T \cup \mathbb{N})^*$ et $w \in (T \cup N)^+$

4. **Langages récursivement énumérables** ou sans restriction dans tous les autres cas.

Il est facile de constater, comme le montre la figure 1.1, la relation d'inclusion entre les classes de grammaires (et les langages correspondants). Les plus larges sont celles de type 0 et les plus restrictives sont les grammaires (langages) de type 3.

1.4 Automates finis

Un automate fini (Finite Automaton, or Finite State Machine) est une machine abstraite qui permet, en un temps fini, d'accepter ou de rejeter des mots fournis à son entrée. Sa réponse est alors binaire, *i.e*, il retourne pour chaque chaîne soumise à lui oui (accepte) ou non (rejette). Cette décision est formulée suivant une fonction de transitions qui joue le rôle de son moteur. Cette fonction est définit souvent par une table qui associe les transitions possibles des différents états selon le symbole du mots actuellement présent devant une sorte de tête de lecture. Le processus de reconnaissance démarre à partir d'un état spécial dit initial, et progresse en respectant la dynamique définie par la table de transitions pour signaler une acceptation du mot si la machine s'arrête sur un état dit final ou signaler un rejet dans le cas contraire. Notez qu'un rejet est aussi émis si l'automate se bloque durant le processus de

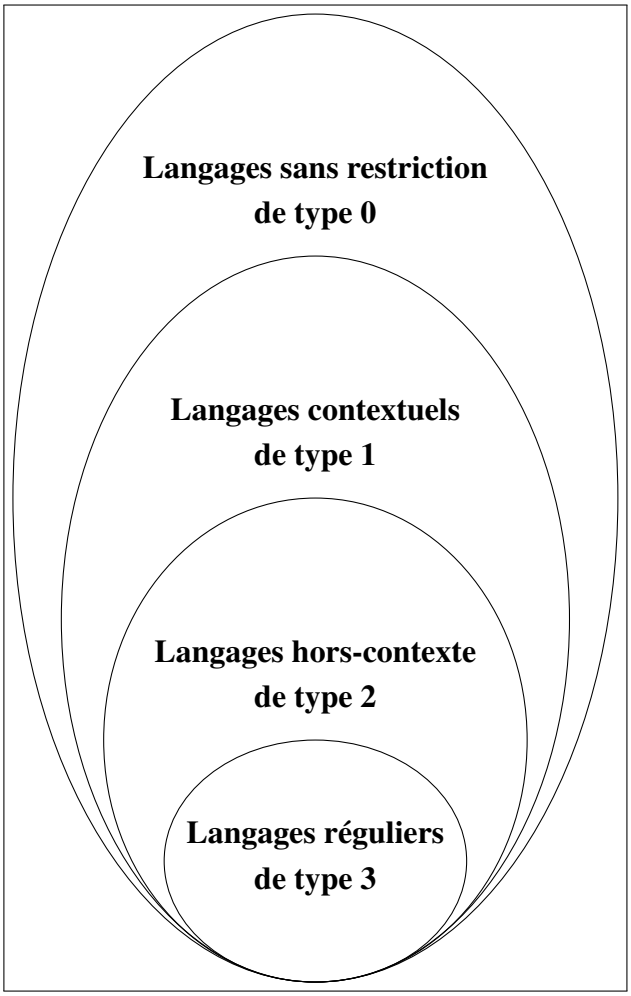


Figure 1.1: Illustration de la hiérarchie de Chomsky

reconnaissance (absence de décision pour une configuration donnée matérialisée par l'inexistence d'une transition par l'état et le symbole actuels). Il existe différents sorts d'automates selon les modalités et les commodités qu'ils offrent.

Les automates sont des modèles de calcul très puissant qui trouve des applications variées dans plusieurs domaines :

- La recherche de motifs : qui exploite l'essence même des automate à savoir reconnaître un motif au sens large.
- Forme une brique de base qui aide dans la synthèse des circuits numériques
- Un outil fondamental de vérification pour plusieurs domaines en particuliers les protocoles de communication
- Différents types d'automates sont utilisés dans la réalisation des compilateurs (analyseurs lexicaux)

Ci-après nous présentons de façon formelle les plus communs des automates finis.

1.4.1 Automates finis déterministes

Un automate fini déterministe **AFD** est un quintuplet $A =< Q, \Sigma, \delta, q_0, F >$ où :

- Q est un ensemble fini d'états.
- Σ est un ensemble fini de symboles dit l'alphabet d'entrée.
- $\delta : Q \times \Sigma \to Q$ est la fonction de transitions. $\delta(p,a) = q$ signifie que de l'état p on passe à l'état q à la lecture du symbole a
- $q_0 \in Q$ est l'état initial
- $F \subseteq Q$ est l'ensemble des états finaux

Un automate est qualifié par **déterministe** (AFD) s'il existe au plus une transition par état et par lettre de l'alphabet. Dans le cas contraire, il est dit non déterministe.

Notons qu'un automate est représenté soit en donnant sa table de transitions qui élucide le lien entre états et symboles de l'alphabet ou sous la forme d'un graphe qui montre explicitement cette fonction de transition.

Exemple 1. *Soit l'automate défini par la donnée des composants suivants :*

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $\delta = \{(q_0, a, q_0), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_2), (q_2, a, q_2), (q_2, b, q_2)\}$
- q_0 *est l'état initial*
- $F = \{q_1\}$

		a	b
$\rightarrow$	q_0	q_0	q_1
$*$	q_1	q_0	q_2
	q_2	q_2	q_2

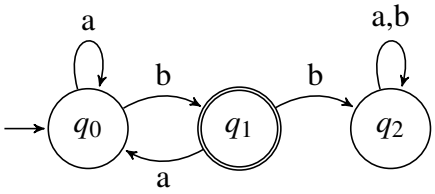


Figure 1.2: Exemple d'un AFD

Configuration d'un automate

Le couple (q, w) représente une configuration de l'automate. où q est un état de l'automate et w un mot de Σ^*. Ainsi (q_0, w) est dit configuration initiale et (q_f, ε) une configuration finale avec $q_f \in F$

Un mot w est accepté par un automate si on peut passer de la configuration initiale de mot w et on termine vers une configuration finale avec le mot vide. C'est à dire s'il existe un chemin qui part de l'état initial et termine dans un des états finaux ou les étiquettes de ses transitions sont les symboles qui composent le mot dans l'ordre. Plus précisément, un automate accepte $w = a_1 a_2 \ldots a_n$ s'il existe une séquence d'états $s_0, s_1, \ldots s_n$ tel que :

- $s_0 = q_0$
- $\delta(s_i, a_{i+1}) = s_{i+1}$
- $s_n \in F$

Dans l'exemple précédent, l'automate accepte bien le mot *abab* mais rejette *abb*. La fonction de transitions δ est aisément étendue au mots de la manière suivante :

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

$$\begin{cases} \hat{\delta}(q,\varepsilon) &= q \\ \hat{\delta}(q,aw) &= \hat{\delta}(\delta(q,a),w) \end{cases}$$

L'ensemble des mots acceptés par un automate forme le **langage** accepté par cet automate. Formellement, on écrit :

$$L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0,w) \in F\}$$

Un automate est **complet** si pour tout état il existe exactement une transition pour chaque symbole de l'alphabet Σ. Deux automates sont **équivalents** s'ils acceptent le même langage.

1.4.2 Automates finis non déterministes

Un automate fini est qualifié de non déterministe (AFN) si sa fonction de transitions autorise le passage d'un état par un symbole vers un ensemble d'états au lieu d'un état unique comme dans le cas déterministe. Autrement dit, un automate non déterministe peut se retrouver sur plusieurs états au même moment. Cette propriété offre une souplesse dans la conception d'automates modélisant un espace très large de problèmes; mais n'améliore pas pourtant son pouvoir de reconnaissance.

Un AFN est donc toujours un quintuplet avec les même composants avec une fonction de transition Δ définie comme suit :

$$\Delta : Q \times \Sigma \to 2^Q$$

Exemple 2. *Soit l'AFN définit ainsi* $A =< \{q_0,q_1,q_2\},\{0,1\},\Delta,q_0,F >$ *avec la fonction* Δ *et l'ensemble* F *schématisés dans la table et la figure ci-dessous :*

		0	1
$\rightarrow$	q_0	$\{q_0,q_1\}$	$\{q_0\}$
	q_1	$\emptyset$	$\{q_2\}$
$*$	q_2	$\emptyset$	$\emptyset$

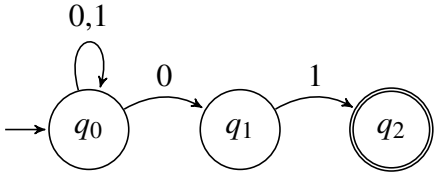


Figure 1.3: Exemple d'un AFN

Le langage d'un AFN est l'ensemble des mots dont l'exécution par l'automate conduit à un ensemble contenant au moins un état final. *i.e* :

$$L(A) = \{w \in \Sigma^* \mid \hat{\Delta}(q_0, w) \cap F \neq \emptyset\}$$

Cette dernière définition n'est correcte qu'en étendant la fonction de transitions aux mots de la manière suivante :

$$\begin{cases} \hat{\Delta}(q, \varepsilon) & = & \{q\}\ \forall q \in Q \text{ cas de base} \\ \hat{\Delta}(q, u) & = & \{p_1, \dots p_k\} \text{ Pour } p_i \in Q,\ a \in \Sigma \text{ et} \\ \bigcup_{i=1}^{k} \hat{\Delta}(p_i, a) & = & \{r_\} \dots r_m\} \text{ alors :} \\ \hat{\Delta}(q, w) & = & \{r_1, \dots r_m\} \text{ et } w = ua \end{cases}$$

Équivalence entre AFDs et AFNs

M.O. Rabin & D.S. Scott introduisent en 1959 [26] un théorème intéressant qui stipule que tout langage accepté par un AFN est aussi accepté par un AFD. Ce résultat donne également un algorithme de passage d'un AFN vers un AFD équivalent dit la construction des sous-ensembles (subset construction).

Théorème 1.4.1. *$\mathscr{A}_\mathscr{N}$ un AFN : Il existe un AFD $\mathscr{A}_\mathscr{D} \mid L(\mathscr{A}_\mathscr{N}) = L(\mathscr{A}_\mathscr{D})$*

Donc à tout AFN $A_N =< Q_N, \Sigma, \Delta, q_0, F_N >$ correspond un AFD $A_D =< Q_D, \Sigma, \delta, q_1, F_D >$ avec :

- $Q_D = 2^{Q_N}$ c-à-d les parties de Q_N sans les états inaccessibles (ne possédant pas un chemin de l'état initial)
- $q_1 = \{q_0\}$
- δ(S,a) = $\bigcup_{p \in S} \Delta(p, a)$ avec $S \in Q_N$
- $F_D = \{S \in Q_N \mid S \cap F_N \neq \emptyset\}$

Exemple 3. *L'application de cette construction à l'AFN de l'exemple 2 précédent à la Figure 1.3 donne l'AFD ci-dessous dans la Figure 1.4:*

		0	1
$\rightarrow$	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*$	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

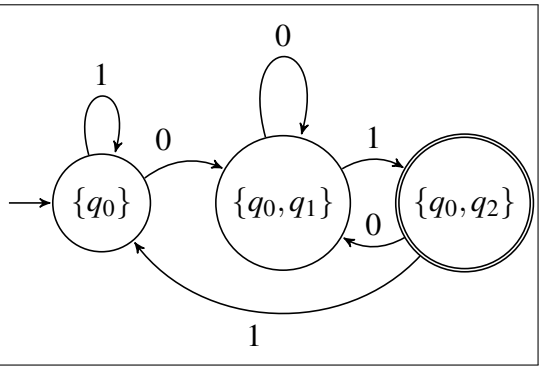


Figure 1.4: AFD obtenu par la construction des sous-ensemble appliquée à l'AFN précédent

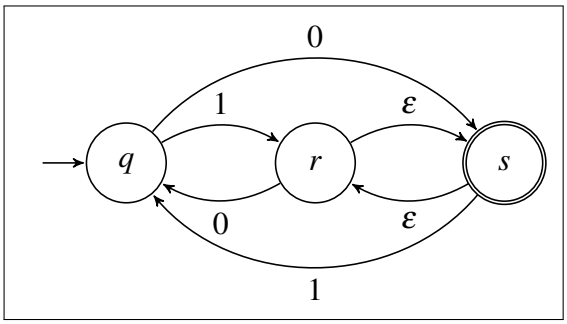


Figure 1.5: Exemple d'un ε-AFN

1.4.3 Automates finis non déterministes avec ε-transitions

Une ε-transition dite aussi spontanée ou instantanée est une transition qui ne consomme pas de symbole de l'entrée, elle est effectuée sur le mot vide ε. Ce type de transitions est une sorte d'indéterminisme et constitue une commodité qui offre une souplesse dans la conception des automates mais n'améliore pas la puissance de ces derniers. Formellement, la définition ajoute seulement à la fonction de transitions des entrées pour ε.

Un ε-AFN est un quintuplet incluant les mêmes composants :
$A_\varepsilon =< Q, \Sigma, \Delta_\varepsilon, q_0, F >$ avec : $\Delta_\varepsilon : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$

L'acceptation d'un mot est toujours l'existence d'un chemin de l'état initial vers un état final. Par exemple, l'ε-AFN de la Figure 1.5 accepte 001 mais pas 01.

Comme pour les AFNs, à tout ε-AFN, il existe un AFN sans ε-transitions équivalent. Cette transformation est basée sur la notion de ε-fermeture (ou clôture).

Théorème 1.4.2. *Soit $\mathscr{A}_\varepsilon$ un ε-AFN, alors il existe un AFN sans ε-transitions $\mathscr{A}_\mathscr{N}$*

tel que $L(\mathscr{A}_\varepsilon) = L(\mathscr{A}_\mathscr{N})$

ε-clôture

l'ε-clôture d'un état p, noté $C_\varepsilon(p)$, est l'ensemble de tous les états atteignables à partir de p par zéro, un ou plusieurs ε-transitions. Formellement de façon récursive la fermeture en ε est spécifiée comme suit :

$$\begin{cases} p & \in & C_\varepsilon(p) \\ \text{si } q \in C_\varepsilon(p) \;\&\; q' \in \Delta_\varepsilon(q,\varepsilon) & \mathit{Alors} & q' \in C_\varepsilon(p) \\ C_\varepsilon(E) & = & \bigcup_{s\in E} C_\varepsilon(s) \end{cases}$$

Dans l'ε-AFN précédent, nous avons $C_\varepsilon(q) = \{q\}$; et $C_\varepsilon(r) = C_\varepsilon(s) = \{r,s\}$

Élimination des ε-transitions

Plusieurs applications requièrent des automates sans transitions vides. Il existent plusieurs algorithmes pour rendre un automates avec ε-transitions libre de celles-ci. La procédure la plus directe est similaire à celle de la déterminisation. Il faut seulement prendre en compte les ε-transitions par le mécanisme des ε-clôtures. Le résultat de cette dernière procédure est un automate fini déterministe.

En général, on peut concevoir un algorithme qui élimine les transitions spontanées et produit un automate non déterministe. Voici la procédure de suppressions des transitions nulles d'un état p vers un autre état q :

- Trouver toutes les transitions partantes de q
- Refaire ces transitions (en gardant les mêmes étiquettes) cette fois émanant de p
- Si p est un état initial; rendre q aussi initial
- Si q est un état final; rendre p final aussi.

1.5 Expressions régulières

Les expressions régulières ou rationnelles sont un autre moyen pour dénoter des langages. Elles représentent l'équivalent algébrique des automates finis. Voir la section 2.5 pour les techniques de passage entre ces expressions et les automates.

Si r est une expressions régulière $L(r)$ est le langage dénoté ou spécifié par l'expression r. Trois opérateurs dit réguliers sont utilisés pour former ces expressions : l'union (+), la concaténation (.) et l'itération ou l'étoile de Kleene (*).

Une définition récursive des expressions régulières sur une alphabet Σ est la suivante :

- $\emptyset$ est une expression et $L(\emptyset) = \emptyset$
- ε est une expression régulière et $L(\varepsilon) = \{\varepsilon\}$
- Pour tout $a \in \Sigma$: a est une expression régulière et $L(a) = \{a\}$
- Si r et s sont deux expressions régulières, alors :
  1. $r+s$ est une expressions régulière et $L(r+s) = L(r) \cup L(s)$
  2. $r.s$ ou simplement rs est une expressions régulière et $L(rs) = L(r)L(s)$
  3. r^* est une expressions régulière et $L(r^*) = (L(r))^*$
  4. (r) est une expressions régulière et $L((r)) = L(r)$. cette dernière clause n'est pas un opérateur, mais permet de définir des expressions régulières parenthésées.

Dans les expressions complexes, il faut tenir compte de l'ordre de priorité des opérateurs réguliers. Voici cet ordre du plus fort vers le plus faible :

1. () : les parenthèses
2. * : l'étoile de Kleene
3. . : la concaténation
4. + : l'union

Exemple 4. *Ci-après des exemples d'expressions régulières avec les langages associés :*

$$\begin{array}{lcl}
(0+1)1 & : & \{01,11\} \\
0+10^* & : & \{0,1,10,100,1000,\ldots\} \\
(a+b)^* & : & \{\varepsilon,a,b,aa,ab,ba,bb,aaa,\ldots\} \text{ toutes les chaînes sur} \{a,b\} \\
(0(0+1))^* & : & \{\varepsilon,00,01,0000,0001,0100,0101,\ldots\} \text{ les mots sur } \{0,1\} \\
 & & \text{de longueur pair dont 1 est toujours en position pair}
\end{array}$$

Attention à la priorité sus-indiquée. Par exemple les expressions suivantes ne sont pas équivalentes :

$$\begin{array}{lcl}
ab & \neq & ba \\
ab+c & \neq & a(b+c) \\
ab^* & \neq & (ab)^*
\end{array}$$

Pour finir, voici quelques propriétés utiles sur les expressions régulières E, F et G étant des expressions régulière:

$$\begin{array}{lcll}
E+F & = & F+E & \\
E+E & = & E & \\
(E+F)G & = & EG+EF & \\
E(F+G) & = & EF+EG & \\
E+\emptyset & = & \emptyset+E & = \emptyset \\
E\emptyset & = & \emptyset E & = \emptyset \\
E\varepsilon & = & \varepsilon E & = \varepsilon \\
\emptyset^* & = & \varepsilon^* & = \varepsilon \\
E(FE)^* & = & (EF)^*E & \\
(E+F)^* & = & (E^*F)^*E^* & = (F^*E)^*F
\end{array}$$

1.6 Langages réguliers et propriétés de fermeture

Un langage L sur un alphabet Σ est reconnaissable s'il est accepté par un automate fini. Il est qualifié de régulier ou rationnel s'il est dénoté par une expression régulière.

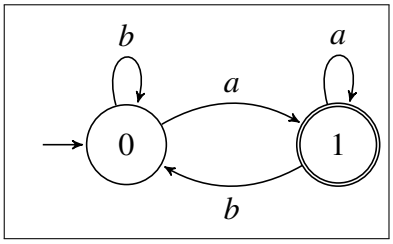


Figure 1.6: Un AFD qui reconnaît le langage dénoté par l'expression $(a+b)^*a$

Par exemple, le langage sur $\{a,b\}$ des mots se terminant par un a est reconnaissable (reconnu par l'automate de la Figure 1.6) et aussi régulier (dénoté par l'expression régulière suivante : $(a+b)^*a$).

S.C. Kleene [18] introduit le théorème d'équivalence entre automates finis et expressions régulières qui porte son nom. Il prouva que la classe des langages reconnaissables est égale à la classe des langages réguliers. La preuve donne une méthode pour passer d'un automate à une expressions régulière équivalente et inversement. Cette preuve introduit pour le premier sens une construction dite de Thompson. Le deuxième sens est possible via le lemme d'Arden. Afin d'alléger ce support ne nous détaillons pas ces constructions ici. Pour de plus ample information, veuillez vous référez aux slides du cours et les références bibliographiques.

En résumé les modèles présentés dans ce chapitre sont équivalent : tout ce que peut être reconnu par un AFD l'est aussi par : un AFN, ε-AFN, une expression régulière et évidement une grammaire régulière.

Théorème 1.6.1. *Les automates finis, les expressions régulières et les grammaires linéaires spécifient la même classe de langages (les langages réguliers)*

La classe des langages réguliers jouit de quelque propriétés dite de fermeture. Autrement dit, des opérations sur des éléments de cette classe qui préserve la régularité et produit un langage de la même classe. Nous les citons ici sans démonstration :

Théorème 1.6.2. *La famille des langages réguliers et fermée par :*

- *Union*
- *Intersection*
- *Concaténation*

- *Itération*
- *Complémentation*
- *Différence*
- *Transposé*
- *Homomorphisme*

Enfin, notons qu'il existe des langages qui ne sont pas réguliers. Le lemme de pompage est un outil qui valide la régularité d'un langage, veuillez vous référer au référence bibliographiques pur plus de détail [16].

Dans la partie analyse lexicale nous utiliserons les langages réguliers, par contre nous nous recourons aux langages hors contextes dans la partie analyse syntaxique. Les langages contextuels trouvent une application dans l'analyse sémantique.

Exercices du chapitre 1

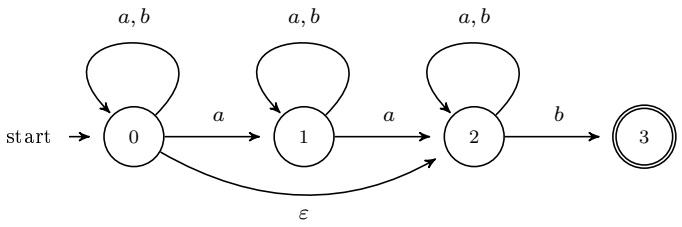
RAPPEL

Exercice 1

$L = \{ab, aa, baa\}$ un langage. Lesquels des mots suivants sont dans L^* :
abaabaaabaa, *aaaabaaaa*, *baaaaabaaaab*, *baaaaabaa*

Exercice 2

Soit l'AFN $\mathcal{A}$ suivant :



(a) Indiquer tous les chemins étiquetés par *aabb*

(b) $\mathcal{A}$ accepte-t-il *aabb*

(c) Donner sa table de transition

(d) Déterminiser $\mathcal{A}$

Exercice 3

Construire un automate fini qui accepte le langage L des mots sur $\{a, b\}$ ayant exactement deux a et plus de deux b. Modifier cet automate de manière à accepter le langage $\overline{L}$. Trouver des constructions pour accepter : L^2, $L^2 \setminus L$.

Exercice 4

Concevoir un AFN ayant :
— Trois états pour $\{ab, abc\}^*$
— Au plus cinq états pour la langage $\{abab^n \mid n \geq 0\} \cup \{aba^n \mid n \geq 0\}$
— Un seul état final pour $\{a\} \cup \{b^n \mid n \geq 1\}$

Exercice 5

Donner un AFD pour chacun des langages sur $\{0, 1\}$ suivants :
— Tout 00 est suivi immédiatement par un 1 (01, 0010,0010011001 sont dans ce langage et 00100 et 0001 ne le sont pas)
— Les mots où le symbole le plus à gauche est différent de celui le plus à droite.
— Le langage dénoté par l'ER : $aa^* + aba^*b^*$

Exercice 6

Donner les grammaires sur $\{a, b\}$ qui génèrent les langages suivants :

— Les mots avec exactement un seul a
— Les mots ayant au moins un a
— Les mots avec au plus trois a
— $L_1 = \{a^n b^m \mid n \geq 0, m > n\}$
— $L_2 = \{a^n b^{2n} \mid n \geq 0\}$
— $L_3 = L_1 L_2$
— $L_4 = L_1 \cup L_2$
— $L_5 = \{w \mid |w|\%3 = 0\}$
— $L_6 = \{ww^R \mid w \in \{a, b\}^+\}$

Exercice 7

Soit l'expression régulière suivante : $(a^*bc^+ + aacb^+)^+$

(a) Donner une grammaire régulière à droite qui engendre le même langage

(b) Déduire l'AFD équivalent

(c) Analyser les chaînes *aabcaacb* et *bcacb*

Exercice 8

Utiliser la construction de Thompson pour concevoir un AFN correspondant à l'expression : what|who|why, puis donner l'équivalent déterministe et minimal.

Exercice 9

Donner une grammaire régulière à gauche qui engendre le langage formé de 0 et de 1 contenant au moins une séquence 010 ou une séquence 000, puis déduire l'AFD équivalent et analyser les chaînes 0110101 et 01101100.

Exercice 10

Considérons le langage sur $\{a, b\}$ formé de mots n'ayant pas deux caractères consécutifs identiques.

(a) Ce langage est-t-il régulier ? justifier

(b) Trouver une expression rationnelle qui le dénote

Exercice 11

Décrire les langages dénotés par les expressions rationnelles suivantes :

1. $a(a+b)^*a$
2. $((\varepsilon + a)b^*)^*$
3. $(a+b)^*a(a+b)(a+b)$
4. $a^*ba^*ba^*ba^*$

Exercice 12

Minimiser l'automate ci-dessous défini par sa table de transitions (0 est l'état initial et 2 l'unique état final). Donner l'ER du langage reconnu par cet automate.

	a	b
$\rightarrow 0$	1	5
1	6	2
$\leftarrow 2$	0	2
3	2	6
4	7	5
5	2	6
6	6	4
7	6	2

CHAPTER 2

Analyse lexicale

2.1 Rôle de l'analyseur lexical

L'analyseur lexical (scanner, lexer ou encore tokenizer en anglais) lit le texte du programme source de gauche à droite [1] caractère par caractère afin d'identifier les différentes entités lexicales qu'il comporte [4, 10]. Pour cela, il découpe le code source pour reconnaître des sous chaînes (dites lexèmes) ayant un rôle dans le langage de programmation du code à compiler et détermine pour chacune sa catégorie ou classe. Le couple (lexème, classe) est appelé token.

La suite des tokens trouvée par l'analyseur lexical est alors communiquée à l'analyseur syntaxique qui représente la phase suivante du processus de compilation. Cette communication est initiée par l'analyseur syntaxique qui demande à chaque fois le token suivant de l'analyseur lexical. Cette interaction est illustrée dans la Figure 2.1.

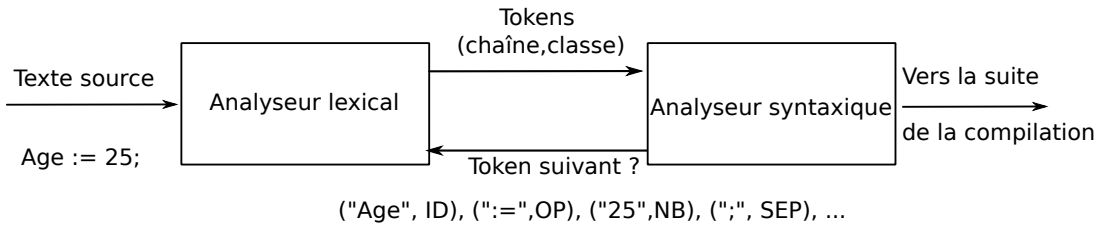


Figure 2.1: Interaction entre l'analyses lexicale et syntaxique

Pour chaque unité lexicale on associe [4] :

- Une **lexème** : la chaîne de caractères correspondante dans le code source

[1] Les langages sont issus du latin. Un scan de droite à gauche pourrait être envisagé dans les cas de l'arabe par exemple

```
if (moyenne >= 10) {
  Credits = 5;
} else {
  Credits = 0;
}
```

Figure 2.2: Portion d'un code source fourni à l'analyseur lexical

```
if (moyenne >= 10) {\n\tCredits = 5;\n} else {\n\t Credits = 0;\n}
```

Figure 2.3: Le code précédent vu par l'analyseur lexical comme un flot de caractères

- Une **classe** : la catégorie à laquelle appartient cette unité parmi plusieurs définies par le langage source
- Un **motif** : un modèle permettant de décrire l'unité lexicale
- des **attributs** : pour certaines types d'unités lexicales, toute information complémentaire pour la caractériser (type, valeur, position dans le code source, etc.)

Une implémentation d'un analyseur lexical devrait donc prendre en charge les tâches ci-dessous :

1. Reconnaître les sous-chaînes et leurs classes (les différents tokens).
2. Interagir avec la table de symboles pour la gestion des unités lexicales (codage, insertion, recherche et mise à jour éventuelle dans la suite du processus de compilation)
3. Éliminer l'ensemble de parties inutiles dans le code, telles que les commentaires et les blancs.
4. Signaler les erreurs lexicales, ou dans les implémentations actuelles entamer une phase de récupération en supprimant une partie de l'entrée jusqu'à ce que la reprise de l'analyse soit possible, ou en proposant des corrections à ces erreurs (insérer, supprimer, permuter des caractères par exemple), voir la Section 3.6.

```
<Mot-clé, "if"> <Séparateur, "("> <Identificateur, "moyenne">
<Opérateur, ">="> <Nombre, "10"> <Séparateur, ")"> <Séparateur, "{">
<Identificateur, "Credits"> <Opérateur, "="> <Nombre, "5">
<Séparateur, ";"> <Séparateur, "}"> <Mot-clé, "else"> <Séparateur, "{">
<Identificateur, "Credits"> <Opérateur, "="> <Nombre, "0">
<Séparateur, ";"> <Séparateur, "}">
```

Figure 2.4: Suite de tokens émis par l'analyseur lexical

2.2 Table des symboles

La table des symboles est une structure de données centrale pour les compilateurs. Elle sert à consigner de manière incrémentale selon l'avancement du processus d'analyse les tokens reconnus et les éventuels attributs associés. Cette structure offre au moins les directives d'ajout, de mise à jour et de recherche des unités lexicales.

Une table de symbole peut être implémentée par une simple structure linéaire comme une table ou une liste linéaire chaînée. Toutefois, la complexité des constructions des langages de programmation et les tailles des codes manipulés souvent exigent d'autres implémentations plus efficaces. Pour répondre à ces défis une table de symboles peut être implémentée via des structures composées pour la gestion des portées par exemple. Aussi, les arbres binaires de recherche et les tables de hachages sont alors exploitées dans ces structures.

2.3 Spécification lexicale

Tout langage de programmation est construit d'un ensemble de mots (unités lexicales) qu'on peut regrouper en plusieurs catégories. Voici les plus communes.

2.3.1 Classes des unités lexicales

Un langage de programmation typique inclut par exemple les catégories ci-dessous :

1. **Mots clés** : begin, if, then,...
2. **Identificateurs** : i, somme1, age,...
3. **Nombres** (entiers ou flottants) : 14, -65, 12.33e-4,...

4. **Opérateurs** : +,-,*,/,<,<=,>,>=,==,!=,&&,...
5. **Séparateurs** : ;,{, }, ,...
6. **Blancs** : " ", tabulation, nouvelle ligne, ...

Une classe d'unités lexicales englobe donc un ensemble de chaînes obéissant à un motif précis. Pour les catégories en haut on peut lister les motifs suivants :

1. **Mots clés** : la liste des mots réservés du langage.
2. **Identificateurs** : les chaînes non vides formées de lettre et de chiffres commençant par une lettre.
3. **Nombres** : les chaînes non vides de chiffres pour les entiers, et un motif plus élaborés pour les réels (voir plus loin)
4. **Opérateurs** : l'ensemble des symboles représentant les différents opérateurs : arithmétiques, logiques, relationnels, etc.
5. **Séparateurs** : l'ensemble des symboles représentant les caractères utilisés comme séparateurs : le point-virgule, les accolades, les parenthèses, etc.
6. **Blancs** : les chaînes non vide de espace, tabulation et nouvelle ligne.

2.3.2 Expressions régulières notation étendue

Au niveau lexical, la question est de reconnaître des unités lexicales représentées par des chaînes de caractères (finies). Il est claire que ces ensembles forment des langages réguliers, pour lesquels on peut spécifier des expressions régulières. L'analyse lexicale se base sur ces expressions qui forment un outil élégant pour dénoter des ensembles de mots. Elles utilisent les trois opérations régulières par ordre croissant de priorité ($+$: l'union, . : la concaténation et $*$ pour l'itération). Veuillez se référer à la Section 1.5 pour un rappel. Nous donnons ci-après quelque exemples.

Exemple 5. *1.* $(0+1)1 = \{01, 11\}$

2. $0^* = \bigcup_{i>=0} 1^i = \{\varepsilon, 1, 11, 111, \ldots\}$

3. $0^* + 1^* == \{\varepsilon, 0, 00, 000, \ldots, 1, 11, 111, \ldots\}$

4. $(0+1)^* = \bigcup_{i>=0}(0+1)^i = \{0,1,00,01,10,11,000,\ldots\}$ *Toutes les chaînes de* 0 *ou* 1
5. $(0(0+1))^* = \{\varepsilon,00,01,0000,0001,0100,0101,\ldots\} = \{\omega \in \{0,1\}^* \mid |\omega| = 2k, k \in N$ *et 1 en positions paires*$\}$
6. $a+a(a+b)^*a$ *: Les mots sur* $\{a,b\}$ *commençants et finissants par un a*

Exercice L'expression régulière $(0+1)*1(0+1)*$ est équivalente à quelle(s) expression(s) parmi les suivantes :

1. $(01+11)*(0+1)*$
2. $(0+1)*(10+11+1)(0+1)*$
3. $(1+0)*1(1+0)*$
4. $(0+1)*(0+1)(0+1)*$

Depuis les trois opérations régulières de Kleene [18], plusieurs autres extensions et implémentations ont été proposées afin de simplifier et améliorer l'usage des expressions régulières. Voici, ci-dessous, les notations les plus fréquentes [4, 21].

Pour une exploration des expression régulières dans des langages variés tel que : Java, PHP, Python, etc., le lecteur est invité à consulter le livre [30].

Exercice

1. Donner l'expression régulière des nombres réels en notation scientifique (signe et partie décimale optionnelle) [21] :
   - $[-+]?[0-9.]+$ reconnaît plus 1.2.3.4
   - $[-+]?[0-9]+\backslash.?[0-9]+$ reconnaît peu: rejette .12 et 12.
   - $[-+]?[0-9]*\backslash.?[0-9]+$ n'accepte pas 12.
   - $[-+]?[0-9]+\backslash.?[0-9]*$ idem pour .12
   - $[-+]?[0-9]*\backslash.?[0-9]*$ accepte le vide et le point seul.

   Solution : $[-+]?([0-9]*\backslash.?[0-9]+|[0-9]+\backslash.)([eE][+-]?[0-9]+)?$
   Une expression régulière pour les identificateurs pourrait être :

$$[a-zA-Z][a-zA-Z0-9]*$$

Table 2.1: Expressions régulières étendues communes

$A\|B$	exprime l'union ou l'alternative. Équivalente à $A+B$
A^+	au moins une occurrence de A. Équivalente à AA^*
$A?$	0 ou une occurrence de A. Équivalente à $A\|\varepsilon$, exprime l'option
a	le caractère a
$[abc]$	un caractère parmi a, b ou c. Mettre les caractères - et] en premier pour les inclure dans cette expression. Les méta-caractères n'ont aucune signification ici sauf les caractères précédés d'un \
$[\hat{}abc]$	tout caractère autre que a, b ou c sauf fin de ligne
.	tout caractère sauf fin de ligne
$\backslash a$	le caractère a littéralement. utilisée pour les métacaractères (les caractères ayant un rôle dans les expressions régulières comme : \, | , (,), [, { , $, ^,*, + ,?)
"s"	la chaîne s littéralement
$\hat{}r$	l'expression r mais uniquement en début de ligne
r\$	l'expression r mais uniquement en fin de ligne
$r\{m\}$	m occurrences de r. Si un nom est met à l'intérieur des accolades, elle référence une expression ayant le même nom.
$r\{m,n\}$	de m à n occurrences de r
$r\{n,\}$	au moins n occurrences de r
r/s	l'expression r lorsqu'elle est suivie de s
()	utilisés pour regrouper des expressions régulières

2. Donner une expression régulière pour dénoter des adresses électroniques du genre : quelquun@serveur.université.xy

2.4 Reconnaissance des unités lexicales

Après avoir spécifié les différentes unités lexicales d'un langages en utilisant des expressions régulières, la question maintenant est de vérifier si une chaîne s appartient au langage ainsi défini dénoté par ces expressions.

2.4.1 Quelque problèmes dans la reconnaissance des tokens

Avant de voir l'approche de reconnaissance, nous présentons ci-dessus quelque difficultés inhérentes à l'analyse lexicale dans des exemples de langages de

programmation. Le but est de cerner les problèmes qui peuvent compliqués la tâche des scanners [6].

1. **Fortran**

   Les blancs sont non significatifs. Il en résulte que par exemple l'identificateur $val1$ et $v\ al\ 1$ sont identiques. Par ailleurs, l'analyseur lexical ne peut distinguer les deux instructions suivantes qu'après avoir lu le (. ou la ,).

   - DO 5 I = 1,25
   - DO 5 I = 1.25

   Ce genre de problème nécessite l'anticipation (lookahead en anglais) pour décider du partitionnement opportun des unités lexicales.

2. **PL/I**

   Les mots clés dans cet ancien langage ne sont pas réservés. Ceci dit, un programmeur peut coder :
   if else then then = else; else else = then

3. **C^{++} :**

   Des conflit peuvent surgir avec l'usage des templates imbriqués et l'opérateurs de flux ».

4. **Python :**

   Les blocs sont définis par indentations, ce qui exige de mémoriser les augmentations et retraits de celles-ci.

2.4.2 Approche de reconnaissance

L'analyseur lexical scanne l'entrée de gauche à droite caractère par caractère afin de la partitionner à ses tokens. L'algorithme suivant est adopté [6].

1. Écrire les expressions régulières pour les lexèmes de chaque classe (mots clés (R_{kw}), identificateurs (R_{id}), nombres (R_{nb}),...)

2. L'expressions régulière reconnaissant toute les lexèmes du langage est donc l'union de celles-ci.
   $R = R_{kw} + R_{id} + R_{nb} + \ldots$

3. Pour le préfixe $a_1 \dots a_i$ $(1 \leq i \leq n)$, où n est la taille de l'entrée, vérifier si $a_1 \dots a_i \in L(R)$

4. Si succès (oui), alors la chaîne $a_1 \dots a_i \in L(R_k)$ pour un certain k; sinon déclarer le préfixe comme une unité lexicale erronée.

5. Supprimer $a_1 \dots a_i$ et recommencer à 3.

2.4.3 Conventions

Il est aisé de constater des ambiguïtés dans l'approche précédente. Voici quelque une et les solutions adoptées.

- **Quel préfixe prendre ?**

  Autrement dit à quelle position il faut s'arrêter pour déclarer la détection d'une unité lexicale. Durant le partitionnement, il arrive que $a_1 \dots a_i$ et $a_1 \dots a_j$ avec $j > i$ sont deux préfixes de notre langage. Dans de pareils cas, la convention est de prendre toujours le préfixe le plus long. Cette convention est appelée la règle du *maximal-munch* [27]. Exemples : 325.66e-4 est un seul token au lieu de : 325 suivi de .66, puis e et enfin -4. Les opérateurs doubles sont aussi des exemples : <= (inférieur ou égal aulieu de inférieur puis égal), >=, ==, := , <>, etc.

- **Quelle classe de tokens choisir ?**

  Il est souvent possible de tomber sur un préfixe $a_1 \dots a_i$ qui peut appartenir à plus d'une classe $a_1 \dots a_i \in L(R_m)$ et $a_1 \dots a_i \in L(R_n)$ avec $m \neq n$. L'exemple courant est celui des mots clés qui sont aussi des identificateurs. Dans ces situations, on définit un ordre de priorité entre les classes du langage, et on opte pour celle ayant la plus haute priorité. Dans les implémentations (voir la section 4) l'ordre d'apparition dans la spécification des classes définit cette priorité.

- **Aucune classe ne s'applique !**

  Si le préfixe en cours n'appartient pas au langage, il faut signaler une erreur. Pour éviter le blocage de l'analyseur lexical, il est commode d'intercepter ces cas dans une classe **erreur** ayant la plus faible priorité.

2.5 Implémentation d'analyseurs lexicaux

Après avoir acquis le fondement de l'analyse lexicale d'un code source, son implémentation peut être réalisée par plusieurs méthodes. Nous décrivons dans cette section les plus courantes.

2.5.1 Analyseur lexical en dur

Cette méthode repose sur l'écriture manuelle (ad-hoc) de l'analyseur. On prépare l'ensemble des unités lexicales à reconnaître et on écrit le code de reconnaissance pour chacune d'elles. Le squelette du scanner sera composé essentiellement d'une boucle de lecture guidée par une succession de tests sous la forme d'une pile de if then else ou par un switch case par catégorie de tokens. La simplicité et l'efficacité de cette méthode est évidente. Cependant, elle reste exposée à l'erreur et difficile à maintenir ou à étendre. L'Algorithme 1, donne un exemple d'un analyseur simple permettant de reconnaître les trois tokens : **end**, **else**, et des **identificateurs**.

Algorithme 1 Pseudo code d'un mini-analyseur lexical codé en dur

```
c = car_suiv();
if (c =='e'){
  c =  car_suiv();
  if ( c == 'n'){
    c =  car_suiv();
    if ( c == 'd'){
      c =  car_suiv();
      if ( ! lettre(c) && ! chiffre(c))
        {
        return KW_END;
        }
      else return ID;}
    else return ID;}
  else if(c == 'l'){
//... continuer pour reconnaitre else ou un ID
}
```

2.5.2 Implémentation d'automates finis

Ici, on construit l'automate (déterministe et minimal) [16] global de reconnaissance de l'ensemble des unités lexicales à partir de leur expressions régulières les spécifiant. L'automate (union des automates des classes de ces unités) est en suite

implémenté soit itérativement ou par des procédures (ou fonctions selon le langage utilisé) mutuellement récursives associées aux différents états de l'automate (lente dû à la récursion entre appels de modules). Un pseudo-code est montré dans l'Algorithme 2.

Algorithme 2 Implémentation d'un automate par des fonctions

```
int etat0(void) {
  switch(fgetc(input)){
    case car1 : return etat1(); break;
    case car2 : return etat2(); break;
    ...
    case EOF :  return final(); break;
    default : return 0;
    }
}
```

Une autre approche d'implémentation d'un automate consiste à déclarer et initialiser sa table de transitions. La reconnaissance est essentiellement une boucle de lecture et de passages entre états. Cette méthode est relativement efficace si la taille de la table des transitions reste raisonnable. Un pseudo-code général de reconnaissance d'une chaîne par un automate via sa table de transitions est présenté dans l'Algorithme 3.

Algorithme 3 Implémentation d'un automate fini via sa table de transitions

```
  etat = e0;
  lexeme = ''
  c = fgetc(input);
  while (c != EOF && etat != ERREUR)
    {
      etat = TTrans[etat][c];
      c = fgetc(input);
      lexeme += c
    }
  return (etat != ERREUR && final(etat));
}
```

Il est à rappeler que la passage d'une expression régulière en l'automate équivalent peut être effectué par plusieurs méthodes. Nous donnons ici la construction de Thompson qui génère un automate non déterministe (avec ε-transitions) et la méthode des dérivées permettant d'avoir directement un automate déterministe.

2.5.2.1 Construction de Thompson

Cette construction [31] suit la définition récursive des expressions régulières et donne pour chacun des six cas l'automate associé. Les automates des cas de base sont schématisés dans la Figure 2.5, et ceux des cas d'induction dans la figure 2.6.

La Figure 2.7, quant à elle, montre l'automate obtenu après application de la

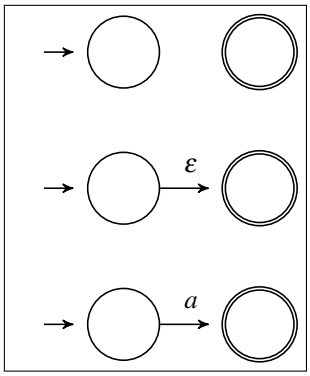


Figure 2.5: Construction de Thompson pour les cas de base : ϕ, ε et a

construction de Thompson correspondante à l'expression $(a|b)*c$.

2.5.2.2 Dérivées

La dérivée (dûe à Janusz A. Brzozowski) [8, 25] d'une expression régulière E sur un alphabet Σ par rapport à un symbole $a \in \Sigma$, notée $a^{-1}(E)$ est définie comme suit :

$$\begin{cases} a^{-1}(\phi) = a^{-1}(\varepsilon) = a^{-1}(b) & = \phi \text{ pour } b \neq a \\ a^{-1}(a) & = \varepsilon \\ a^{-1}(E+F) & = a^{-1}(E) + a^{-1}(F) \\ a^{-1}(EF) & = a^{-1}(E)(F) + \varepsilon(E)a^{-1}(F) \\ a^{-1}(E*) & = a^{-1}(E)(E*) \end{cases}$$

Avec $\varepsilon(E)$ est une fonction qui aide dans le calcul des dérivées. Elle exprime l'appartenance ou non du mot vide à E.

$$\varepsilon(E) = \begin{cases} \varepsilon & \text{si } \varepsilon \in E \\ \phi & \text{sinon} \end{cases}$$

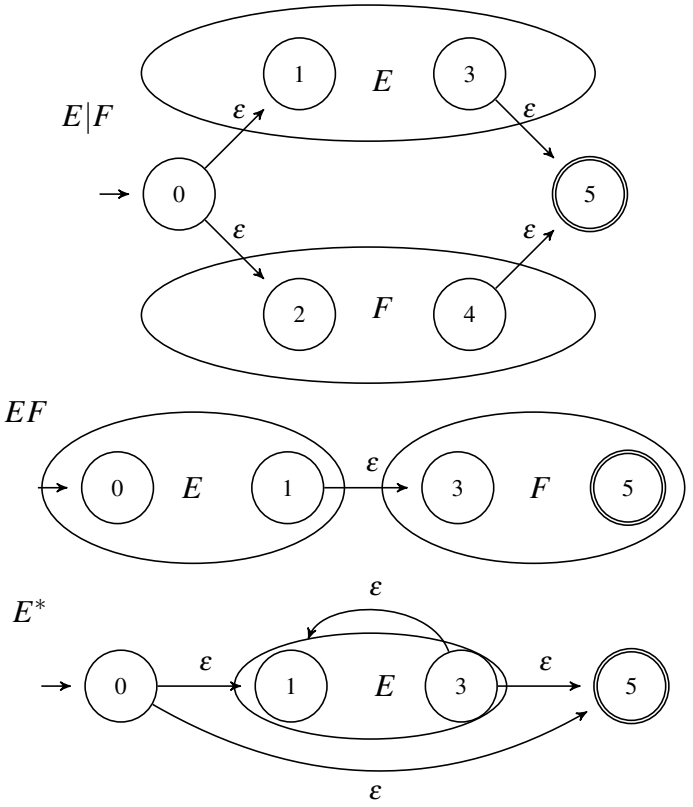


Figure 2.6: Construction de Thompson pour les cas d'induction : $E|F$, EF et E^*

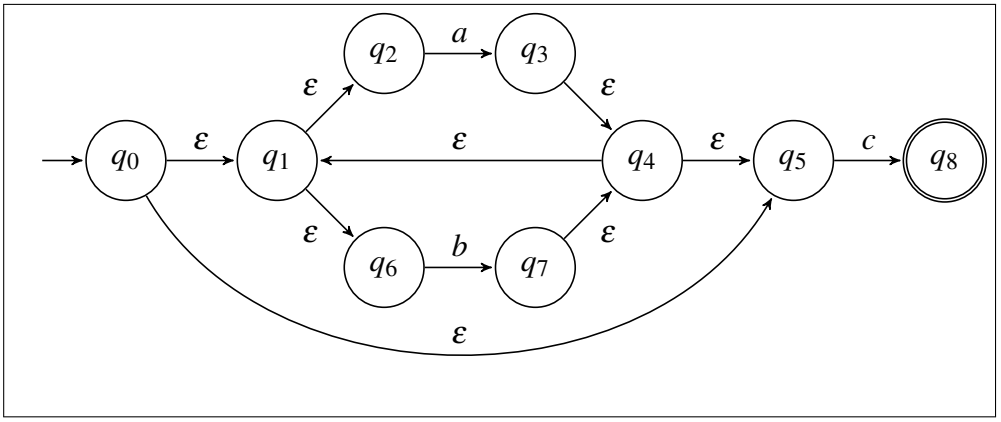


Figure 2.7: Construction de Thompson pour l'expression $(a|b)*c$

Intuitivement, la dérivée d'un langage dénotée par l'expression régulière E par rapport à un symbole a est l'ensemble des mots générés en supprimant le premier a des mots de E qui commencent par a. Par exemple, la dérivée de ab^* par rapport à a est b^*, par contre sa dérivée par rapport à b est le langage vide.

Les états de l'automate résultat sont les différentes dérivées. Une transition, par le symbole a est crée entre un état correspondant à l'expression s et celui correspondant à l'expression dérivée $a^{-1}(s)$. Un état est final si l'expression qu'il représente inclut le mot vide ε. Le lecteur intéressé pourra trouver les détails et les preuves (correction, terminaison) de cette construction dans [8]. Un exemple de cette construction pour l'expression $(a+b)*bab$ est montré dans la Figure 2.8

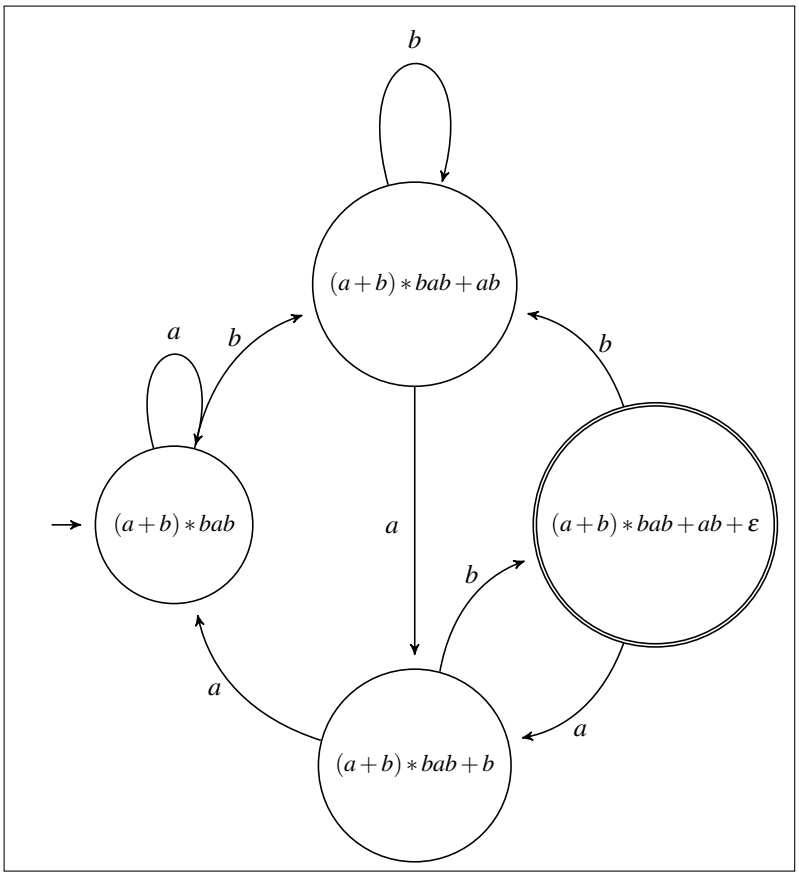


Figure 2.8: Automate de $(a+b)*bab$ par la méthode des dérivés.

2.5.3 Générateurs d'analyseurs lexicaux cas de Flex

La maîtrise des outils et des modèles théoriques inhérents à la compilation a permis la mise en œuvre de générateurs de différents types d'analyseurs pour la plupart des environnements et langages de programmation. Un générateur d'analyseur lexical est un outil qui accepte en entrée la spécification des unités lexicales sous formes

d'expressions régulières et produit en sortie le code permettant la reconnaissance de ces unités. Le cœur de ce code est l'automate (déterministe et minimal) global de reconnaissance. Parmi ces outils, nous citons : lex d'unix [24] et sa version libre de gnu sous linux (Flex) [3], Jlex pour Java [2] et PLY pour Python [1], etc. Dans cette section, nous présentons l'outil Flex.

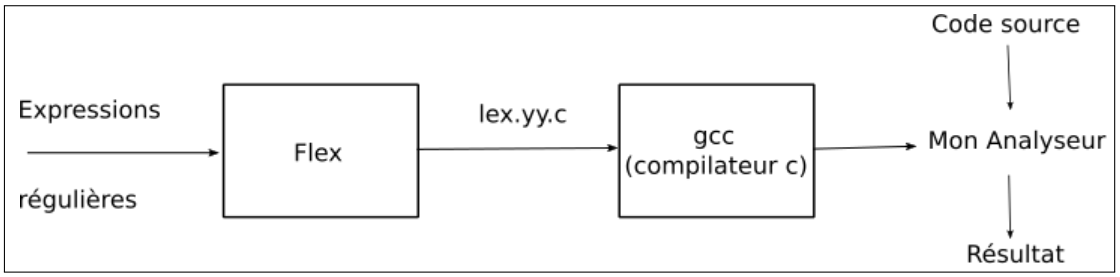


Figure 2.9: Fonctionnement de Flex

2.5.3.1 Structure du fichier de spécification flex

Flex [21], dont le principe est exhibé par la Figure 2.9, génère un code en C (dans le fichier lex.yy.c par défaut où est définie la routine yylex) à partir d'un fichier texte de spécification lexicale (souvent ayant l ou lex comme extension). Le code produit est à compiler pour obtenir l'executable de l'analyseur. Le fichier d'entrée de flex est composé des parties ci-dessous comme l'illustre l'Algorithme 4.

Algorithme 4 Structure générale d'un code Flex

```
%{
  Declaration de variables, constantes, includes, etc.
  /* partie optionnelle qui sera copiee au debut du code source
    produit */
%}
  Declaration des definitions regulieres   (optionnelle aussi)
%%
  Regles de traduction
%%
  Bloc principal et fonctions auxiliaires
  /* cette partie est optionnelle et sera copiee a la fin du code
    source produit */
```

2.5.3.2 Définitions régulières

Elles associent des noms à des expressions qui permettent en suite de simplifier l'écriture du code. Elles ont la forme suivante :
nom *exp*
Voici des exemples :
lettre [a-zA-Z]
chiffre [0-9]
Une expression précédemment définie est ensuite utilisée dans la partie suivante en l'entourant entre une paire d'accolades {}.

2.5.3.3 Règles de traduction

C'est la partie principale d'un code flex. Elle définie des expressions régulières (partie gauche) et les actions associées (partie droite en C) à effectuer dans le cas de leur reconnaissance selon la forme ci-dessous.

```
exp_1   {action_1}
exp_2   {action_2}
...
```

Exemple 6.
```
(0|1)+    { printf("%s est un nombre binaire",yytext);}
```

Un exemple complet de l'utilisation de flex pour générer un analyseur d'un mini-lanage est présenté dans la Figure 2.10. Mentionnons pour finir que Flex offre plusieurs fonctions et variables :

- **yylex()** est la fonction de l'analyseur,
- **yytext** la chaîne reconnue dan le texte source lors de l'action en cours,
- **yyleng** représente la longueur de la chaîne reconnue.
- **ECHO** action par défaut de flex,
- **yyin et yyout** fichier d'entrée/de sortie de flex (stdin et stdout par default).

```
%{
  /* pour la fonction atof() */
  #include <math.h>
%}
NB     [0-9]
ID     [a-z][a-z0-9]*
%%
{NB}+ { printf( " Un Entier : %s (%d)\n", yytext,atoi( yytext ) );}
{NB}+"."{NB}* {printf(" Un flottant : %s (%g)\n", yytext,atof(
    yytext));}
if|then|begin|end|procedure|function { printf( " Mot clef : %s\n",
    yytext );}
{ID}  printf( " Un identificateur : %s\n", yytext );
"+"|"-"|"*"|"/"   printf( " Un Operateur : %s\n", yytext );
"{"[\^{$\;$}}\n]*"}"  /* ignorer les commantaires sur une ligne */
[ \t\n]+ /*       ignorer les espaces et les blanc */
.      printf( " Caractere invalide : %s\n", yytext );
%%
main( argc, argv )
int argc;
char **argv;
{
  if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
  else
    yyin = stdin;
  yylex();
}
```

Figure 2.10: Un exemple complet de code Flex

2.5.3.4 Autres usages de Flex

On peut faire plusieurs tâche une fois une chaîne suivant un motif donné a été reconnue. Cela permet d'exploiter Flex pour effectuer non seulement de l'analyse lexicale, mais aussi des actions variées telles que : recopier, supprimer, convertir, etc.

Exercices du chapitre 2

ANALYSE LEXICALE

Exercice 1

Soit la portion du code source suivante :

```
x = 0 ;\n\twhile  (x  < 10) { \n\tx++;\n }
```

Déterminer le nombre d'unités lexicales pour chaque classe de tokens (Mots-clés, Identificateurs, Nombres, Opérateurs, Blancs, Sep, etc.)

Exercice 2

Soit le code C^{++} ci-dessous

```
float limitedSquare(float x) {
/* return x-squared, but never more than 100 */
return (x <= -10.0 || x >= 10.0)? 100 : x*x; }
```

(a) Diviser ce code à ses tokens appropriés

(b) À quels lexèmes doit-t-on associer des valeurs ? Quelles sont ces valeurs ?

Exercice 3

Proposer des spécifications de l'heure donner au format 12H exemple : "05 :14 PM". Les minutes sont toujours des nombres à deux chiffres, mais l'heure peut être un chiffre unique.

Exercice 4

Donner des expressions régulières pour dénoter :

(a) Une chaîne quotée en C (exemple : "compilation"), sachant qu'une chaîne quotée ne peut s'étendre sur plusieurs lignes

(b) Un commentaire en C^{++} introduit par //

(c) Un commentaire multi-lignes en C^{++} autorisant ou non le caractère * dans le texte du commentaire

Exercice 5

Un commentaire en PASCAL est une suite de 80 caractères au plus délimitée par (* et *) ne contenant pas la chaine *) à moins que celle-ci n'apparaisse dans une chaîne bien quotée.

(a) Construire un automate déterministe qui accepte un commentaire en PASCAL

(b) Écrire un algorithme de reconnaissance d'un commentaire

Exercice 6

Écrire un analyseur lexical qui reconnaît les unités X, Y et Z suivantes. On supposera que les unités du langage sont séparées par un ou plusieurs blancs.

(a) L'unité X est une suite de chiffres décimaux qui peut commencer par le caractère '-'. Sa valeur est inférieur à 32768.

(b) L'unité Y commence par une lettre suivie par une suite de lettres, de chiffres et de tirets. Elle ne se termine jamais par un tiret et ne comporte pas de tirets qui se suivent. Sa longueur maximale est de 31 caractères.

(c) L'unité Z est une suite de chiffres qui peut commencer par le caractère '-' et qui comporte le point décimal. Le nombre de chiffres est inférieur ou égal à 9.

Exercice 7

En SQL, les mots clés et les identificateurs ne sont pas sensibles à la casse. Écrire un programme Flex qui reconnait les mots clés SELECT, FROM, WHERE (avec n'importe quelle combinaison de minuscules et de majuscules), ainsi que l'unité lexicale ID.

Exercice 8

Considérons la chaîne *abbbaacc*. Quelle(s) spécification(s) lexicale(s) produi(sen)t la tokenisation : $ab/bb/a/aac$:

$b+$	$c*$	$a(b\|c*)$	ab
$ab*$	$b+$	$b+$	$b+$
$ac*$	ab		$ac*$
	$ac*$		

Exercice 9

Pour les entrées ci-après énumérées, donner la liste des tokens reconnus par un analyseur implémentant la spécification suivante :

$a(ba)*$
$b*(ab)*$
abd
$d+$

1. *dddabbabab*
2. *ababddababa*
3. *babad*
4. *ababdddd*

Exercice 10

(a) Étant donné la spécification lexicale ci-dessous.

```
%%
aa          printf("1");
b?a+b?      printf("2");
b?a*b?      printf("3");
.           printf("4");
```

Indiquer pour les chaînes suivantes la sortie de l'analyseur et les tokens produits :

— `bbbabaa`
— `aaabbbbbaabanalyse`

(b) Donner un exemple d'une entrée pour laquelle cet analyseur produit la sortie **123**, ou justifier, le cas échéant, l'inexistence de tel exemple.

(c) Quelle sera pour les mêmes chaînes données en (1) la sortie de l'analyseur, si on remplace seulement la partie expression régulière de la dernière règle par `.+`

CHAPTER 3

Analyse syntaxique

3.1 Rôle de l'analyseur syntaxique

La mission de l'analyseur syntaxique (parser en anglais) est de vérifier la conformité de la suite de tokens délivrés par l'analyseur lexical à la définition syntaxique du langage source souvent donnée par une grammaire à contexte libre (de type 2 dans la hiérarchie de Chomsky). Autrement dit, valider que l'ordre des tokens peut être produit par la grammaire du langage.

Pour les programmes bien formés, *i.e* syntaxiquement corrects, l'analyseur syntaxique construit un arbre syntaxique et le transmet aux phases suivantes du processus de compilation. Dans le cas contraire, il doit signaler une erreur syntaxique et entamer une procédure de rattrapage [4, 13].

Deux approches sont utilisées pour vérifier la validité et construire l'arbre syntaxique d'un programme source. La première adopte une méthode descendante qui consiste à démarrer à partir de l'axiome de la grammaire pour aboutir au code source en effectuant une suite de dérivations; c-à-d des remplacements successives de membres gauches de productions par des membres droits correspondants. La deuxième procède inversement et est qualifiée d'ascendante, car elle part du texte source et remonte à l'axiome par des suites valides de réductions ou de remplacements cette fois de membres droits de production par des membres gauches associés. Ces deux approches seront discutées dans ce chapitre. Commençons par rappeler quelques notions utiles pour la suite du cours.

3.2 Spécification syntaxique

Au niveau de l'analyse lexicale, les lexèmes des tokens, étant des chaînes simples, sont décrits alors par des expressions régulières car ils forment un langage régulier. Le niveau syntaxique des langages de programmation inclut de nombreuses

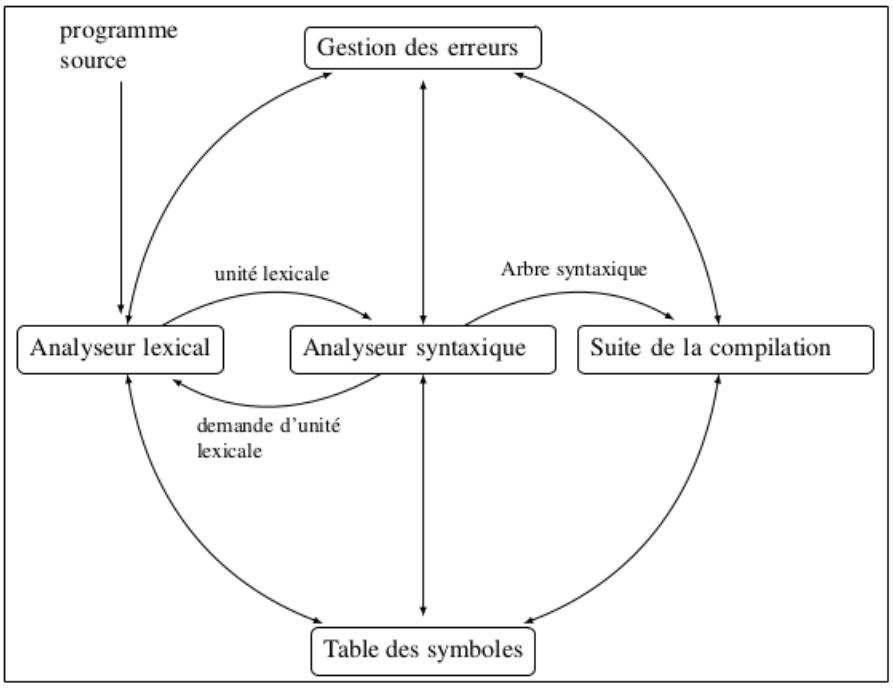


Figure 3.1: Interaction entre les différents modules de la partie analyse d'un compilateur

constructions complexes qui ne sont pas régulières. Ces constructions ne peuvent être décrites par des expressions régulières. Pour l'analyse syntaxique des langages de programmation, nous devons opté pour un autre modèle ou outil plus puissant qui est les grammaires.

Exemple

- $L = \{(^n)^n \mid n \geq 0\}$ langage des expressions bien parenthésées. $(), (())$, tel que dans $((1+2)*3)\ldots$
- Délimiteurs de blocs $\{\ldots\}$ de Java et C ou les **begin end** de Pascal.
- Les suites de **if then else**

Pour que l'analyseur syntaxique puisse détecter les suites valides d'unités lexicales de celles invalides, il a besoin de :

- Un outil décrivant les suites valides d'unités lexicales : les **grammaires**
- Une méthode ou un **algorithme** pour assurer cette séparation.

3.2.1 Grammaires hors contexte et dérivation vs réduction

Les grammaires à contexte libre (non contextuelles, hors contexte ou encore algébriques) sont adaptées à la description de la plupart des constructions des langage de programmation.

Rappelons que les règles de production ou de réécriture d'une grammaire à contexte libre sont de la forme : $A \rightarrow \alpha$, où A est un non terminal $\in \mathbb{N}$ et $\alpha \in (\mathbb{T} \cup \mathbb{N})^*$ une chaîne quelconque de terminaux et de non terminaux. La suite de tokens fournie par l'analyseur lexical forme une chaîne représentant la forme du texte source vu au niveau syntaxique. La question ici est de montrer qu'une telle chaîne disons w appartient bien au langage généré par la grammaire G en question. Autrement dit, vérifier si la proposition : $w \in \mathbb{L}(\mathbb{G})$? est vraie.

La réponse à cette question peut être donnée par deux principales méthodes : la première est descendante : en partant de l'axiome de la grammaire et opérant une succession de substitutions dite **dérivations** (remplacement du membre gauche d'une production ici toujours un non terminal par le membre droit) jusqu'à aboutir à la chaîne. Cette méthode est à l'origine des méthode d'analyse syntaxique descendante. Une alternative consiste à adopter le chemin inverse. En commençant à partir de la chaîne du texte source, nous effectuons cette fois-ci une séquence de **réductions** (remplacement du membre droit d'une règle de production par le membre gauche) de proche en proche jusqu'à arriver à l'axiome. Cette méthode donne naissance aux méthodes d'analyse syntaxique ascendantes que nous allons traité ultérieurement.

Dans un processus de dérivation, si toujours le non terminal le plus à gauche (resp. à droite) est remplacé en premier en parle de **dérivation la plus à gauche (DPG)** (resp. à droite (DPD)). **Leftmost** vs **rightmost** derivation en anglais.

Exemple 7. *Soit la grammaire ci-dessous :*

$$\begin{cases} S \rightarrow aTb \mid c & \text{et la chaîne } w = accacbb \\ T \rightarrow cSS \mid S \end{cases}$$

DPG: $S \rightarrow aTb \rightarrow ac\underline{S}Sb \rightarrow acc\underline{S}b \rightarrow acca\underline{T}bb \rightarrow acca\underline{S}bb \rightarrow accacbb$
DPD: $S \rightarrow a\underline{T}b \rightarrow acS\underline{S}b \rightarrow acSa\underline{T}bb \rightarrow acSa\underline{S}bb \rightarrow ac\underline{S}acbb \rightarrow accacbb$

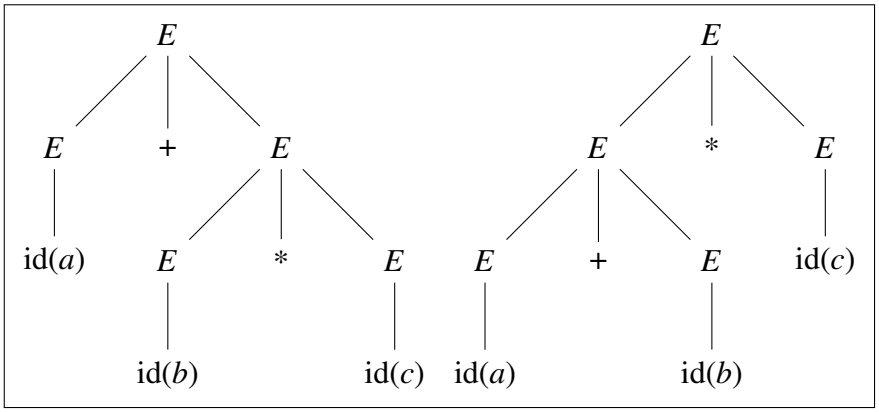


Figure 3.2: Deux arbres syntaxiques pour dériver la chaîne : $w = a + b * c$

Exemple 8.

$$E \to E + E \mid E * E \mid id \qquad \textit{et la chaîne : } w = a + b * c$$

Avant d'entamer les méthodes d'analyse syntaxique, nous discutons dans la section suivante quelque propriétés des grammaires qui peuvent influencer la qualité ou voire même l'existence d'une telle méthode d'analyse. Pour un analyseur syntaxique efficace, la grammaire doit possède certaines propriétés.

3.2.2 Qualités et formes particulières des grammaires

3.2.2.1 Ambiguité

Pour une grammaire G, s'il existe un mot w de $L(G)$ ayant plusieurs arbres syntaxiques, on dit que G est **ambiguë**. Nous pouvons rencontré l'ambiguïté dans plusieurs cas. Voici deux exemples typiques de grammaires ambiguës indispensables dans tous les langages de programmation.

- Les expressions arithmétiques :

$$E \to E + E \mid E * E \mid (E) \mid id$$

- Les expressions conditionnelles : (I et C sont deux non terminaux permettant

de générer les instructions et les conditions respectivement)

$$\begin{cases} I \rightarrow if\ C\ then\ I \\ I \rightarrow if\ C\ then\ I\ else\ I \end{cases}$$

Exemple 9. *if* $x > 10$ *then if* $y < 0$ *then* $a := 1$ *else* $a := 0$

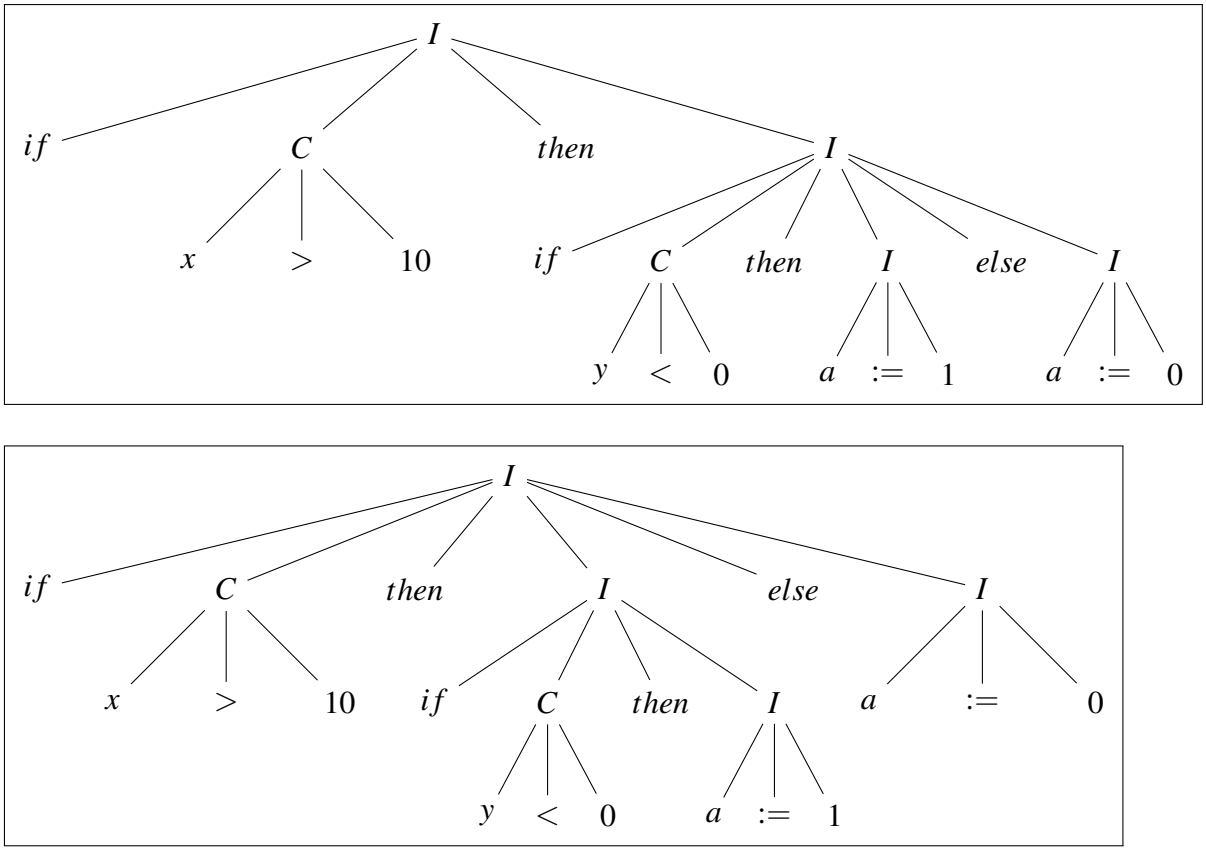


Figure 3.3: Deux arbres syntaxiques de la même chaîne dans la grammaire des instructions conditionnelles

L'ambiguïté est une propriété indésirable dans la construction d'analyseurs syntaxiques. Malheureusement, il n'existe aucune méthode ni pour détecter, ni pour transformer l'ambiguïté d'une grammaire [4, 6, 16]. Comment faire ?

Devant une grammaire ambigue, nous devons :

- Soit la transformer, *i.e* construire une grammaire équivalente non ambiguë

- Soit la retenir mais introduire en plus des mécanismes de désambiguïsation. On peut fixer par exemple des règles de priorité, ou définir des conventions de précédence ou d'associativité (à gauche ou à droite), etc.

La grammaire ambiguë des expressions arithmétiques précédente peut être transformée comme suit :

$$E \to T + E \mid T \qquad T \to id * T \mid id \mid (E) * T \mid (E)$$

En utilisant la grammaire transformée il n'y a qu'une seule façon de dériver la chaîne de l'exemple celle de l'arbre syntaxique à gauche dans la figure 3.2.

3.2.2.2 Récursivité à gauche

Dans les langages de programmation les productions sont souvent définies en terme d'elles mêmes. par exemple la production utilisée dans les déclarations d'une liste de variables :

$$L \to v \mid L, v$$

Une grammaire est dite récursive à gauche (RG) de façon directe si elle contient une règle de la forme :

$$A \to A\alpha \mid \beta \qquad A \in \mathbb{N}; \alpha, \beta \in (\mathbb{T} \cup \mathbb{N})^*$$

Parfois la RG n'est pas explicite ou apparente. Une grammaire est dite récursive gauche de façon indirecte s'il existe une règle de la forme :

$$A \to B\alpha \mid \beta \qquad A, B \in \mathbb{N} \qquad B \to^+ A\gamma \mid \Gamma \qquad \alpha, \beta, \gamma, \Gamma \in (\mathbb{T} \cup \mathbb{N})^*$$

On peut définir de la même façon la récursivité à droite.

Certaines méthodes d'analyse syntaxique ne peuvent fonctionner avec une grammaire récursive à gauche (l'analyse descendante), ce qui nécessite une transformation. Le schéma de l'élimination de la RG repose sur l'introduction d'un non terminal intermédiaire qui prend en charge la récursivité mais à droite en commençant par générer la chaîne la plus à gauche.

$$A \to A\alpha \mid \beta \Leftrightarrow \begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{cases}$$

Dans le cas général, on procédera de la même manière :

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta \mid \ldots \mid \beta_m \Leftrightarrow \begin{cases} A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \end{cases}$$

Exemple 10. *Récursivité gauche directe:*

$$\begin{cases} S \to aS \mid b \mid Ac \\ A \to Aa \mid c \end{cases} \Rightarrow \begin{cases} S \to AS \mid b \mid Ac \\ A \to cA' \\ A' \to aA \mid \varepsilon \end{cases}$$

RGI: Faire des substitutions de manière à faire apparaître une RGD puis appliquer les règles:

$$\begin{cases} S \to Ac \mid b \\ A \to Bd \mid a \\ B \to Ac \mid d \end{cases} \Rightarrow \begin{cases} S \to Ac \mid b \\ A \to Acd \mid dd \mid a \\ B \to Ac \mid d \end{cases} \Rightarrow \begin{cases} S \to Ac \mid b \\ A \to Acd \mid dd \mid a \end{cases} \begin{cases} S \to Ac \mid b \\ A \to aA' \mid ddA' \\ A' \to cdA' \mid \varepsilon \end{cases}$$

3.2.2.3 Factorisation

L'analyseur syntaxique lit de gauche à droite la liste des unités lexicales fournies par l'analyseur lexical. Il est souhaitable qu'il puisse décider à la rencontre d'une unité lexicale quelle production doit utiliser. Ceci est impossible si les membres droits des productions ont les mêmes préfixes.

L'idée de la factorisation est de réécrire la grammaire de sorte à différer l'indéterminisme jusqu'à avoir lu suffisamment d'unités lexicales de l'entrée pour pouvoir faire le bon choix.

Exemple 11.

$$\begin{aligned} I \to\ & \textit{if } C \textit{ then } I \textit{ else } I \\ \mid\ & \textit{if } C \textit{ then } I \end{aligned}$$

Ici. On préfère réécrire la grammaire :

$$I \to \textit{if } C \textit{ then } I\ T$$
$$T \to \textit{else } I \mid \varepsilon$$

En général,

$$A \to aX \mid aY \mid aZ \mid \alpha \mid \beta \qquad \Rightarrow \qquad \begin{cases} A \to aB \mid \alpha \mid \beta \\ B \to X \mid Y \mid Z \end{cases}$$

Exemple 12 (Factorisation)**.** *Soit la grammaire*

$$\begin{cases} A \to abC \mid aBd \mid aAD \\ B \to bB \mid \varepsilon \\ C \to d \mid \varepsilon \\ D \to a \mid b \mid \varepsilon \end{cases} \Rightarrow \begin{cases} A \to aX \\ X \to bC \mid Bd \mid AD \\ B \to bB \mid \varepsilon \\ C \to d \mid \varepsilon \\ D \to a \mid b \mid \varepsilon \end{cases} \Rightarrow$$

$$\begin{cases} A \to aX \\ X \to bC \mid bBd \mid d \mid AD \\ B \to bB \mid \varepsilon \\ C \to d \mid \varepsilon \\ D \to a \mid b \mid \varepsilon \end{cases} \Rightarrow \begin{cases} A \to aX \\ X \to bY \mid d \mid AD \\ Y \to C \mid Bd \\ B \to bB \mid \varepsilon \\ C \to d \mid \varepsilon \\ D \to a \mid b \mid \varepsilon \end{cases}$$

3.2.2.4 Grammaire ε-libre

Une grammaire est ε-libre si aucune production ne contient ε sauf l'axiome, qui ne doit pas apparaître en partie droite d'aucune production.

$$G = \begin{cases} S \to a \mid A \\ A \to AB \mid \varepsilon \\ B \to aAbB \mid b \end{cases} \Leftrightarrow \begin{cases} S \to a \mid \varepsilon \\ A \to AB \mid B \\ B \to aAbB \mid abB \mid b \end{cases}$$

Les production vides posent souvent des difficultés dans l'analyse syntaxique; les isoler est en général un remède aux difficultés qu'elles génèrent.

3.2.2.5 Grammaire sous la forme normale de Chomsky

Dans une grammaire sous cette forme normale, toute les règles sont de la forme

$$\begin{cases} A \to BC & A,B,C \in \mathbb{N} \\ A \to a & a \in \mathbb{T} \\ S \to \varepsilon & \end{cases}$$

Les arbres de dérivation des grammaires sous la forme normale de Chomsky sont binaires ce qui peut offrir une souplesse dans l'analyse.

3.2.2.6 Grammaire sous la forme normale de Greibach

Une grammaire est sous forme normale de Greibach si elle est ε-libre et toutes les règles sont de la forme :

$$A \to a\alpha \qquad A \in \mathbb{N}, \alpha \in \mathbb{N}^*, a \in \mathbb{T} \text{ ou } S \to \varepsilon$$

Les productions dans une grammaire sous la forme normale de Greibach commencent toujours par un terminal, ce qui simplifie l'analyse.

3.3 Analyse syntaxique descendante

Dans cette section, nous décrivons les deux principales méthodes d'analyse syntaxique descendante à savoir : la descente récursive et LL. La première est un ensemble de modules mutuellement récursifs alors que la deuxième est itérative et exploite une table dite la table LL.

3.3.1 Analyse par la descente récursive

Dans cette méthode [4, 23], nous associons à chaque non terminal de la grammaire une procédure ou une fonction. L'analyse se fait à l'aide d'appels de procédures. L'analyseur syntaxique est donc un ensemble de procédures mutuellement récursives dont la structure rassemble à celle de la grammaire. Son exécution commence par le lancement de la procédure associée à l'axiome.

Dans la procédure d'un non terminal, nous vérifions l'entrée à la rencontre d'un terminal et appelons les procédures associées aux non terminaux selon leur apparitions dans les règles de productions. Si une discordance est trouvée nous levons une erreur.

Algorithme 5 Procédure Non-terminal N()

Choisir une production $N \rightarrow X_1 X_2 \dots X_n$
Pour i de 1 à n **Faire**
  Si $X_i \in \mathbb{N}$ **Alors**
    appeler la procédure $X_i()$
  Sinon Si $X_i =$ terme courant de la chaîne **Alors**
      avancer()
    Sinon erreur

Exemple 13. *Soit la grammaire augmentée ci-dessous :*

$$G_1 = \begin{cases} Z \rightarrow S\# \\ S \rightarrow cAd \\ A \rightarrow ab|c \end{cases}$$

Analysons par cette méthode la chaîne : cabd#. Le déroulement du code de la Figure 3.4 est montré dans la Table 3.1.

Table 3.1: Exemple d'analyse par la descente récursive

Pile des appels	chaîne
Z	cabd#
ZS	abd#
ZSA	bd#
ZS	d#
Z	# ⇒chaîne acceptée

Exemple 14. *Donner un analyseur en utilisant la descente récursive de la*

```
Z(){          /*  Programme pricipal de l'analyseur */
  S()
  si tc = '#' alors
    Accepter()
  sinon erreur()
}
S(){            /* procedure associee a S */
  si tc='c' alors {
    Avancer()
    A()
    si tc = 'd' alors
      Avancer()
    sinon erreur()
    }
  sinon erreur()
}
A(){           /* procedure associee a A */
  si tc = 'a' alors {
    Avancer()
    si tc = 'b' alors
        Avancer()
    sinon erreur()
    }
  sinon si tc = 'c' alors
        Avancer()
      sinon erreur()
}
```

Figure 3.4: Code de l'analyseur par descente récursive de G_1

grammaire suivante et analyser la chaîne : bcaa#

$$G_2 : \begin{cases} Z \rightarrow S\# \\ S \rightarrow bSa \mid A \\ A \rightarrow cAa \mid aS \mid \varepsilon \end{cases}$$

Remarques

1. Z joue le rôle du main de l'analyseur
2. *Avancer()* est la fonction permettant de lire (consommer) le token courant et passer au suivant
3. *tc* est le terminal ou le token courant, et # la marque de fin de la chaîne

```
Z(){          /*  Programme pricipal de l'analyseur */
  S()
  si tc = '\#' alors
    Accepter()
  sinon erreur()
}
S(){            /* procedure associee a S */
  si tc='b' alors {
    Avancer()
    S()
    si tc = 'a' alors
        Avancer()
    sinon erreur() }
  sinon A()
}
A(){           /* procedure associee a A */
  si tc = 'c' alors {
    Avancer()
    A()
    si tc = 'a' alors
        Avancer()
    sinon erreur()  }
  sinon si tc = 'a' alors {
        Avancer()
        S()
      }
  sinon ;     /* rien */
}
```

Figure 3.5: Code de l'analyseur par descente récursive de G_2

4. *erreur()* est la routine de gestion des erreurs

5. Méthode simple à implémenter et relativement efficace

6. Elle n'est pas générale, car liée à la grammaire

7. Pour faire une analyse performante déterministe il faut :

   - Éliminer la récursivité gauche afin d'éviter les boucles infinies
   - Factoriser
   - Tenir compte d'autres indéterminismes qui sont liés aux productions notamment celles générant la chaîne vide

Table 3.2: Exemple non concluant d'une analyse par descente récursive d'une chaîne apartenant au langage de la grammaire

Pile des appels	chaîne
Z	bcaa#
ZS	bcaa#
ZSS	caa#
ZSSAA	aa#
ZSSAAS	a#
ZSSAASASA	#
ZSSAASAS	#
ZSSAASA	#
ZSSAASA	#
ZSSAAS	#
ZSSAA	#
ZSSA	# $\Rightarrow$ erreur ? pourtant la chaîne appartient au langage

3.3.2 Analyse LL(k)

Le problème de l'analyse descendante est comment déterminer les productions appropriées afin d'arriver à la chaîne en partant de l'axiome. Dans la méthode LL l'analyseur, en regardant k caractères de la chaîne, prédit par contre à chaque étape la bonne production à appliquer afin d'éviter les retours arrières.

Cette méthode prédictive et non récursive utilise une pile explicite et effectue la recherche dans une table dite **table d'analyse LL(k)** [22, 28]. Souvent $k = 1$ et l'analyse est dite **LL(1)**. Une grammaire est LL(1) si on arrive à accepter une chaîne en regardant un seul symbole de prévision.

LL(k) stands for **l**eft to right scanning perfoming a **l**eftmost derivation using k tokens of lookahead.

Fonctionnement

Nous disposons de la chaîne d'entrée terminée par # et une table d'analyse à deux dimensions. Initialement, la pile contient les symboles de la grammaire avec # au fond au dessous de l'axiome. Les lignes de la table pour les non terminaux, et les colonnes représentent les terminaux avec #. La case à la ligne de N et la colonne de

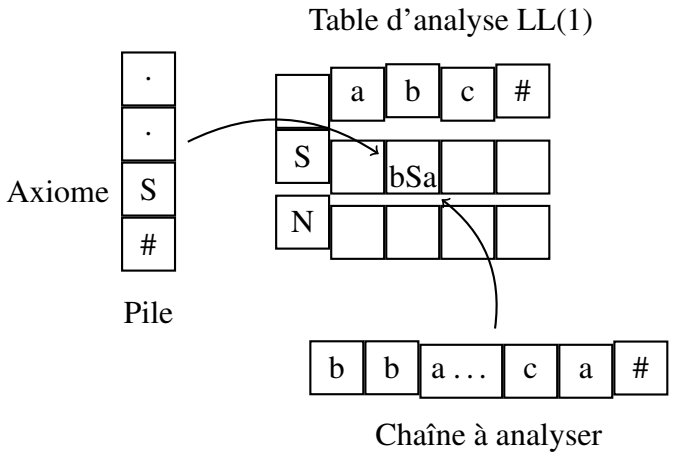


Figure 3.6: Fonctionnement de l'analyse LL

a donne la production à appliquer lorsque *N* est au sommet de la pile et *a* le terminal courant.

En général dans cette méthode, nous avons trois cas :

1. Si le sommet de la pile est un terminal qui coïncide avec le terminal courant : on dépile et on avance dans la chaîne

2. Si le sommet correspond à un non terminal, on consulte la table : si la case à la ligne de ce non terminal et la colonne du terminal courant est non vide : on dépile et on empile le miroir du membre droit de la production trouvée. Si l'entrée est vide, une erreur est signalée.

3. Si la lecture est terminée (terme courant est #), et le sommet est aussi # l'analyse est réussie et terminée.

3.3.2.1 Ensembles DÉBUT et SUIVANT

Afin de construire un analyseur descendant prédictif, on utilise un caractère d'anticipation (lookahead) pour décider de l'action adéquate. Les débuts ou les préfixes de tous les membres droits des production d'un non terminal doivent être disjoints. Par ailleurs, la grammaire doit être exempte de toute forme de récursivité à gauche afin d'éviter les boucles interminables. Nous définissons ici deux concepts qui aident dans la construction de la table d'analyse LL.

DÉBUT

L'ensemble **DEB** constitue la collection de tous les terminaux qui peuvent commencer une chaîne ou ses dérivées. Formellement :

Définition 6.

$$DEB(X) = \{a \in \mathbb{T} \mid X \rightarrow^* a\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\} \text{ Avec } \alpha \in (\mathbb{T} \cup \mathbb{N})^*$$

Pour calculer les ensembles **DEB** d'une chaîne X, appliquer les règles ci-après récursivement jusqu'à stabilité des ensembles.

Algorithme 6 Calcul de l'ensemble DEB(X)

1. Si X est un terminal ou commence par un terminal Ajouter ce terminal à $DEB(X)$
2. Si X est ε ou dérive directement ou indirectement la chaîne vide ajouter ε à $DEB(X)$
3. Si X est un non terminal et $X \rightarrow Y_1 \ldots Y_n$ une production de G avec $Y_i \in (\mathbb{T} \cup \mathbb{N})$:
   - Ajouter $DEB(Y_1)$ à $DEB(X)$
   - Ajouter $DEB(Y_i)$ si $\varepsilon \in DEB(Y_1) \ldots, \varepsilon \in DEB(Y_{i-1})$
   - Ajouter ε à $DEB(X)$ si $\varepsilon \in DEB(Y_1) \ldots, \varepsilon \in DEB(Y_n)$

Il est aisé d'étendre cet ensemble à un préfixe de longueur au plus k (on dit un k-préfixe). Ainsi :

$$DEB_k(X) = \{u|_k \in \mathbb{T}^* \mid X \rightarrow^* u\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\} \text{ Avec } \alpha \in (\mathbb{T} \cup \mathbb{N})^*$$

avec $u|_k$ est le k-préfixe de u. Ce dernier est spécifié, pour un mot $u = a_1 a_1 \ldots a_n$ ($a_i \in \mathbb{T}$) par la définition ci-dessous :

$$w|_k = \begin{cases} a_1 a_2 \ldots a_n & \text{Si } n \leq k \\ a_1 a_2 \ldots a_k & \text{Sinon} \end{cases}$$

SUIVANT

De même, l'ensemble **SUIV** est la collection des terminaux qui peuvent suivre (apparaître immédiatement à droite) un symbole non terminal.

Définition 7.

$$SUIV(X) = \{a \in \mathbb{T} \mid S \rightarrow^* \beta X a \gamma\} \; Avec \; \beta, \gamma \in (\mathbb{T} \cup \mathbb{N})^* \; , \; X \in \mathbb{N}$$

Suivre les règles suivantes pour calculer les ensembles SUIVANT de manière récursive jusqu'à stabilité des ensembles SUIVANT:

Algorithme 7 Calcul de l'ensemble SUIV

1. Ajouter # à $SUIV(S)$ ou S est l'axiome de la grammaire (découle de la production ajoutée $Z \rightarrow S\#$)
2. Si $A \rightarrow \alpha B \beta$: Ajouter $DEB(\beta) \setminus \{\varepsilon\}$ à $SUIV(B)$
3. si $\begin{cases} A \rightarrow \alpha B \\ ou \\ A \rightarrow \alpha B \beta \wedge \beta \rightarrow^* \varepsilon \end{cases}$: Ajouter les $SUIV(A)$ dans $SUIV(B)$

De manière analogue l'ensemble $SUIV$ peut être étendu au suivants d'ordre k comme suit :

$$SUIV_k(X) = \{w \in \mathbb{T}^* \mid S \stackrel{+}{\Rightarrow} \beta X \gamma \text{ et } w \in DEB_k(\gamma\#)\}$$

Dans les exemples suivants, nous calculons les ensembles DEB et SUIV.

Exemple 15.

$$G_3 : \begin{cases} S \rightarrow aAS \mid b \\ A \rightarrow a \mid bSA \end{cases}$$

	DEB	*SUIV*
S	*a,b*	*#,a,b*
A	*a,b*	*a,b*

Exemple 16.

$$G_4 : \begin{cases} S \rightarrow d \mid XYS \\ Y \rightarrow c \mid \varepsilon \\ X \rightarrow a \mid Y \end{cases}$$

	DEB	*SUIV*
S	*a,c,d*	#
X	*a,c,ε*	*a,c,d*
Y	*c,ε*	*a,c,d*

Exemple 17.

$$G_5 : \begin{cases} S \to ABCe \\ A \to aA \mid \varepsilon \\ B \to bB \mid cB \mid \varepsilon \\ C \to de \mid da \mid dA \end{cases}$$

	DEB	*SUIV*
S	*a,b,c,d*	#
A	*a,ε*	*b,c,d,e*
B	*b,cε*	*d*
C	*d*	*e*

Exemple 18.

$$G_6 : \begin{cases} S \to aSb \mid cd \mid SAe \\ A \to aAdB \mid \varepsilon \\ B \to bb \end{cases}$$

	DEB	*SUIV*
S	*a,c*	*a,b,e,#*
A	*a,ε*	*d,e*
B	*b*	*d,e*

Exemple 19.

$$G_7 : \begin{cases} E \to TX \\ X \to +E \mid \varepsilon \\ T \to iY \mid (E) \\ Y \to *T \mid \varepsilon \end{cases}$$

	DEB	*SUIV*
E	(,i	#,)
X	+,ε	#,)
T	(,i	+,#,)
Y	*,ε	+,#,)

Exemple 20.

$$G_8 : \begin{cases} S \to ABCD \\ A \to a \mid \varepsilon \\ B \to CD \mid b \\ C \to c \mid \varepsilon \\ D \to Aa \mid d \mid \varepsilon \end{cases}$$

	DEB	*SUIV*
S	a,b,c,d,ε	#
A	a,ε	a,b,c,d,#
B	a,b,c,d,ε	a,c,d,#
C	c,ε	a,c,d,#
D	a,d,ε	a,c,d,#

Exemple 21.

$$G_9 : \begin{cases} S \to ASSA \mid \varepsilon \\ A \to aSB \mid b \\ B \to bB \mid \varepsilon \end{cases}$$

	DEB	*SUIV*
S	a,ε	a,b,#
A	a,b	a,b,#
B	b,ε	a,b,#

Exemple 22.

$$G_{10} : \begin{cases} S \to ABSb \mid \varepsilon \\ A \to aBb \mid b \\ B \to bB \mid cS \mid \varepsilon \end{cases}$$

	DEB	*SUIV*
S	a,b,ε	a,b,#
A	a,b	a,b,c
B	b,c,ε	a,b

Les résultats suivants donnent les conditions nécessaires et suffisantes pour qu'une grammaire soit $LL(k)$.

Théorème 3.3.1. *Une grammaire $G =< T,N,S,P >$ est $LL(k)$ si et seulement si pour toute pair de productions différentes de P : $A \to \alpha$ et $A \to \beta$ nous avons :*

$DEB_k(\alpha\gamma) \cap DEB_k(\beta\gamma) = \emptyset$ pour tout γ tel que $S \overset{+}{\Rightarrow} \omega A\gamma$

Théorème 3.3.2. *Une grammaire $G =< T,N,S,P >$ est $LL(k)$ si et seulement si pour toute pair de productions différentes de P : $A \to \alpha$ et $A \to \beta$ nous avons :*

$$DEB_1(\alpha) \odot_1 SUIV_1(A) \cap DEB_1(\beta) \odot_1 SUIV_1(\gamma) = \emptyset$$

Où $\odot_k$ est l'opération de concaténation à k-préfixe

Corollaire 3.3.3. *Une grammaire est LL(1) si elle est non récursive à gauche et si pour toute production $A \to \alpha_1 \mid \ldots \mid \alpha_n$, nous avons pour tout $i \neq j$:*

- *$DEB(\alpha_i) \cap DEB(\alpha_j) \neq \emptyset$*
- *Si $\alpha_i \to^* \varepsilon$ alors $\alpha_j \not\to^* \varepsilon$*
- *Si $\alpha_i \to^* \varepsilon$ alors $DEB(\alpha_j) \cap SUIV(A) = \emptyset$*

Exemple 23. *La grammaire ci-dessous est-elle LL*

$$G_{11} : \begin{cases} S \to aAb \mid bS \\ A \to bB \mid \varepsilon \\ B \to b \end{cases}$$

G_{11} n'est pas récursive gauche et elle est factorisée. Cependant les règles de A posent un problème. En effet l'alternative bB commence par un b qui est aussi un suivant de A ($DEB(bB) \cap SUIV(A) \neq \emptyset$). Donc G n'est pas LL.

3.3.2.2 Construction de la table d'analyse LL

Pour l'élaboration de la table d'analyse LL, suivre le pseudo algorithme suivant. Les entrées vides de la table correspondent à des erreurs. La Table 3.3 montre celle associée à la grammaire G_7 de l'Exemple 18.

Algorithme 8 Construction de la table LL

Pour chaque non terminal A de G **Faire**
  Pour chaque production $A \to \alpha$ **Faire**
    Pour chaque $a \in DEB(\alpha) \setminus \{\varepsilon\}$ **Faire**
      T[A,a] $\cup = A \to \alpha$
    Si $\varepsilon \in DEB(\alpha)$ **Alors**
      Pour chaque $a \in SUIV(A)$ **Faire**
        T[A,a] $\cup = A \to \alpha$

Exemple 24. *Construire pour la grammaire G_7 de l'exemple 19 la table d'analyse LL puis analyser la chaîne : i*(i+i)#.*

Table 3.3: Table LL de G_7

	(	+	*	i	)	#
E	TX			TX		
X		$+E$			ε	ε
T	(E)			iY		
Y		ε	$*T$		ε	ε

Exemple 25. *En utilisant la table d'analyse LL vérifier que la grammaire G_{12} suivante n'est pas LL.* $G_{12} : \begin{cases} S \to iEtSR \mid a \\ R \to eS \mid \varepsilon \\ E \to b \end{cases}$

Table 3.5: Table LL de G_{12}

	a	i	b	t	e	#
S	a	iEtSR	b	t	e	
R					eS,ε	ε
E		b				

La table d'analyse étant multi-définie (case $T[R,e]$), cette grammaire n'est pas LL.

3.4 Analyse syntaxique ascendante

Bien que les méthodes d'analyse syntaxique descendantes sont simples et efficaces, il existe hélas des grammaires qui ne peuvent être analysées en utilisant ces méthodes (les grammaires récursives à gauche par exemple). Nous présentons ici des méthodes plus générales et puissantes qui procèdent de façons ascendante (des feuilles en remontant à la racine) dans la construction de l'arbre syntaxique d'un code source afin de vérifier sa syntaxe.

Les méthodes d'analyse syntaxiques ascendantes lisent toujours l'entrée de gauche à droite, elle cherchent, par contre, à trouver une suite de réductions valides qui permettent de transformer le texte source formés d'une séquences de tokens vers l'axiome de la grammaires décrivant le langage de programmation étudié. Ces

Table 3.4: Analyse LL de $i*(i+i)$

Pile	Chaîne	Action
$\#E$	$i*(i+i)\#$	Remplacer E par TX
$\#XT$	$i*(i+i)\#$	Remplacer T par iY
$\#XYi$	$i*(i+i)\#$	Dépiler et avancer
$\#XY$	$*(i+i)\#$	Remplacer Y par $*T$
$\#XT*$	$*(i+i)\#$	Dépiler et avancer
$\#XT$	$(i+i)\#$	Remplacer T par (E)
$\#X)E($	$(i+i)\#$	Dépiler et avancer
$\#X)E$	$i+i)\#$	Remplacer E par TX
$\#X)XT$	$i+i)\#$	Remplacer T par iY
$\#X)XYi$	$i+i)\#$	Dépiler et avancer
$\#X)XY$	$+i)\#$	Remplacer Y par ε (dépiler)
$\#X)X$	$+i)\#$	Remplacer X par $+E$
$\#X)E+$	$+i)\#$	Dépiler et avancer
$\#X)E$	$i)\#$	Remplacer E par TX
$\#X)XT$	$i)\#$	Remplacer T par iY
$\#X)XYi$	$i)\#$	Dépiler et avancer
$\#X)XY$	$)\#$	Remplacer Y par ε (dépiler)
$\#X)X$	$)\#$	Remplacer X par ε (dépiler)
$\#X)$	$)\#$	Dépiler et avancer
$\#X$	$\#$	Remplacer X par ε (dépiler)
$\#$	$\#$	Accepter et arrêter

méthodes dites $LR(k)$ [19] (pour Left to right scanning performing the Reverse of the right most derivations using k symbols of lookahead) réalisent l'inverse d'une suite de dérivations la plus à droite et effectuent deux actions primitives : **Décaler** (empiler un symbole dans la pile) ou **Réduire** (remplacer un membre droit d'une production présent au sommet de la pile par le membre gauche correspondant) (Shift/reduce in english). Elles exploitent donc une pile et un automate déterministe (l'ensemble est en fait un automate à pile) dans la recherche de membres droits candidats (sous-chaînes susceptibles à conduire à une suite valide de réductions). Ces chaînes candidates sont appelées manches ou handles. l'entier k désigne le nombre de caractères de prédiction à utiliser pour effectuer la bonne action. Ce nombre est rarement supérieur à 1, en raison de la complexité impliquée avec l'augmentation de la taille de cette information sur le fonctionnement de l'analyseur. Commençons par décrire la base de ces méthodes en ne regardant aucun symbole de prédiction, c-à-d avec k nul, après avoir examiné un exemple.

Exemple introductif

Avant d'aller en profondeur, donnons un exemple illustratif de cette famille d'approches et le genre de décisions qu'elles sont confrontées à prendre.

Exemple 26.

$$G_{13} : \begin{cases} Z \to E \\ E \to E + T \mid T \\ T \to T * i \mid i \end{cases}$$

Prenons la chaîne de tokens $i+i*i$ générée par la grammaire précédente G_{13} et essayons de montrer les étapes de déroulement de sa réduction en la lisant de gauche à droite. Testons deux scénarios différents afin de montrer la problématique de l'analyse ascendante. La barre verticale indique la position atteinte dans la lecture de la chaîne.

1. $|i+i*i \overset{decaler}{\Rightarrow} i|+i*i \overset{reduire}{\Rightarrow} T|+i*i \overset{reduire}{\Rightarrow} E|+i*i \overset{decaler}{\Rightarrow} E+|i*i \overset{decaler}{\Rightarrow} E+i|*i \overset{reduire}{\Rightarrow} E+T|*i \overset{reduire}{\Rightarrow} E|*i \overset{decaler}{\Rightarrow} E*|i \overset{decaler}{\Rightarrow} E*i| \overset{reduire}{\Rightarrow} E*T| \overset{?}{\Rightarrow}$ *Blocage ! Cette suite d'actions n'a pas réussi à réduire une chaîne qui appartient bel et bien au langage de la grammaire. Réessayons d'une autre manière.*

2. $|i+i*i \overset{decaler}{\Rightarrow} i|+i*i \overset{reduire}{\Rightarrow} T|+i*i \overset{reduire}{\Rightarrow} E|+i*i \overset{decaler}{\Rightarrow} E+|i*i \overset{decaler}{\Rightarrow}$

$E+i|*i \overset{reduire}{\Rightarrow} E+T|*i \overset{decaler}{\Rightarrow} E+T*|i \overset{decaler}{\Rightarrow} E+T*i| \overset{reduire}{\Rightarrow} E+T| \overset{reduire}{\Rightarrow} E|$ *Une réduction réussie !*

3.4.1 Analyse LR(0)

Une méthode *LR* doit implémenter un mécanisme pour reconnaître l'apparition de membres droits de production et de choisir la bonne action à effectuer (décaler /réduire) au temps moment. D. Knuth [19] remarqua que les préfixes possibles (appelés préfixes viables : ceux qui ne dépassent pas les handles) des membres droits des productions d'une grammaire forment un langage régulier, ce qui signifie qu'il est possible de concevoir un AFD pour les reconnaître. Nous introduisons la notion d'item, qui nous aide à la construction de cet automate.

Définition 8 (Manche (poignée ou handle))**.** *Pour une grammmaire G, une dérivation la plus à droite* $Z \overset{+}{\Rightarrow} \gamma' A\omega \Rightarrow \gamma\alpha\omega$ *et une production* $A \rightarrow \alpha$*,* α *est la handle de la partie à doite* $\gamma\alpha\omega$

Définition 9 (Préfixe viable)**.** *Un préfixe viable est un préfixe d'une chaîne résultat d'une dérivation la plus à droite qui n'excède pas (ne va pas au delà) le handle de cette chaîne.*

Définition 10 (Item)**.** *Un item pour une grammaire G est une production dont le membre droit est marqué par un point. Si* $A \rightarrow \alpha\beta \in P$ *alors* $[A \rightarrow \alpha.\beta]$ *est un item.*

Dans cette notation les crochets servent à distinguer les items et la règle de production associée. Généralement à une règle de production correspondent plusieurs items. Par exemple, quatre items sont associés à la production : $E \rightarrow E-T$ qui sont : $[E \rightarrow .E-T]$, $[E \rightarrow E.-T]$, $[E \rightarrow E-.T]$, $[E \rightarrow E-T.]$. La production $A \rightarrow \varepsilon$ quant à elle donne un seul item : $[A \rightarrow .]$. Un ensemble d'items est qualifié de final (terminal) s'il contient un item dans la marque termine le membre droit de l'item (un point à la fin).

L'intuition derrière un item est simple, à titre d'exemple pour l'item $[E \rightarrow E.-T]$: l'analyseur vient juste de reconnaître une chaîne dérivable de E et espère rencontrer le symbole $-$ suivi d'une chaîne dérivable de T.

Afin de construire notre AFD, l'objectif est d'associer items et préfixes viables. Ainsi $[A \rightarrow \alpha.\beta]$ est valide pour un préfixe viable $\phi\gamma$ si et seulement si il existe une dérivation la plus à droite $S \overset{+}{\Rightarrow} \phi Aw \Rightarrow \phi\alpha\beta w$ où w est un mot terminal [13].

Nous donnons une construction d'un AFD à partir des règles de production de la grammaire qui reconnaît les préfixes viables. Les items seront regroupés en ensembles qui forment les états de cet AFD. Définissons les opérations suivantes pour achever cette construction : **fermeture** qui donne les états de l'automate et **successeur** qui en spécifie les transitions entre ces états.

Fermeture d'un ensemble d'items

La fermeture d'un ensemble d'items I est l'ensemble d'items $C(I)$ construit par application de l'algorithme suivant :

Algorithme 9 Fermeture d'un ensemble d'items LR(0)

Entrée : I : Un ensemble d'items LR(0)
Sortie : La fermeture $C(I)$

```
C(I) ← I
Répéter
    Pour tout item [A → α.Bγ] ∈ C(I) Faire
        Pour tout B → β ∈ P Faire
            Si [B → .β] ∉ C(I) Alors
                C(I) ← C(I) ∪ {[B → .β]}
            Fin Si
        Fin Pour
    Fin Pour
Jusqu'à Stabilité de C(I)
Retourner C(I)
```

Successeur δ

Pour tout $X \in (T \cup N)$, si $[A \to \alpha.X\beta]$ est un item de l'ensemble I alors $\Delta(I,X) = C(J)$ où J comprend tous les items $[A \to \alpha X.\beta]$

Algorithme 10 Successeur d'un ensemble d'items LR(0)

Entrée : I : Un ensemble d'items LR(0) et $X \in (T \cup N \cup \{\#\})$
Sortie : La fonction $\delta(I,X)$

```
J ← φ
Pour tout item [A → α.Xβ] ∈ I Faire
    J ← J ∪ {[A → αX.β]}
Fin Pour
Retourner C(J)
```

Exemple 27. *Pour la grammaire G_{13}, nous avons :*
$I_0 = C(\{[Z \to .E]\}) = \{[Z \to .E], [E \to .E + T], [E \to .T], [T \to .T * i], [T \to .i]\}$
$I_1 = \delta(I_0, E) = \{[Z \to E.], [E \to E. + T]\}$
$\delta(I_1, +) = \{[E \to E + .T], [T \to .T * i], [T \to .i]\}$

Construction de l'automate LR(0)

Algorithme 11 Construction de l'automate LR(0)

1. Augmenter les règles de la grammaire par la production $Z \to S$ où S est l'axiome
2. Générer l'ensemble initial d'items correspondant à $C(\{[Z \to .S]\})$
3. Calculer les autres ensembles en utilisant la fonction $\delta(I, X)$ avec I un nouveau ensemble obtenu et $X \in (T \cup N)$ un symbole qui précède la marque dans les ensembles I trouvés.
4. Continuer tant qu'il y a de nouvels ensembles d'items (états)
5. Tous les états sont finaux

La décision (décaler/réduire) dans un analyseur *LR* peut parfois être non déterministe. On peut rencontrer des conflits, où on parle d'état ambiguë ou invalide. deux types de conflit peuvent surgir :

- décaler/réduire : si l'état contient un item final et un autre non final à transition par un symbole terminal
- réduire/réduire : si l'état inclut deux items finals différents.

Lemme 3.4.1. *Une grammaire G est $LR(0)$ si l'automate $LR(0)$ associé ne contient aucun état ambiguë*

Une grammaire LR(0)

La grammaire G_{14} ci-dessous est $LR(0)$, son automate est montré dans la Figure 3.8:

$$G_{14} : S \to aSb \mid c$$

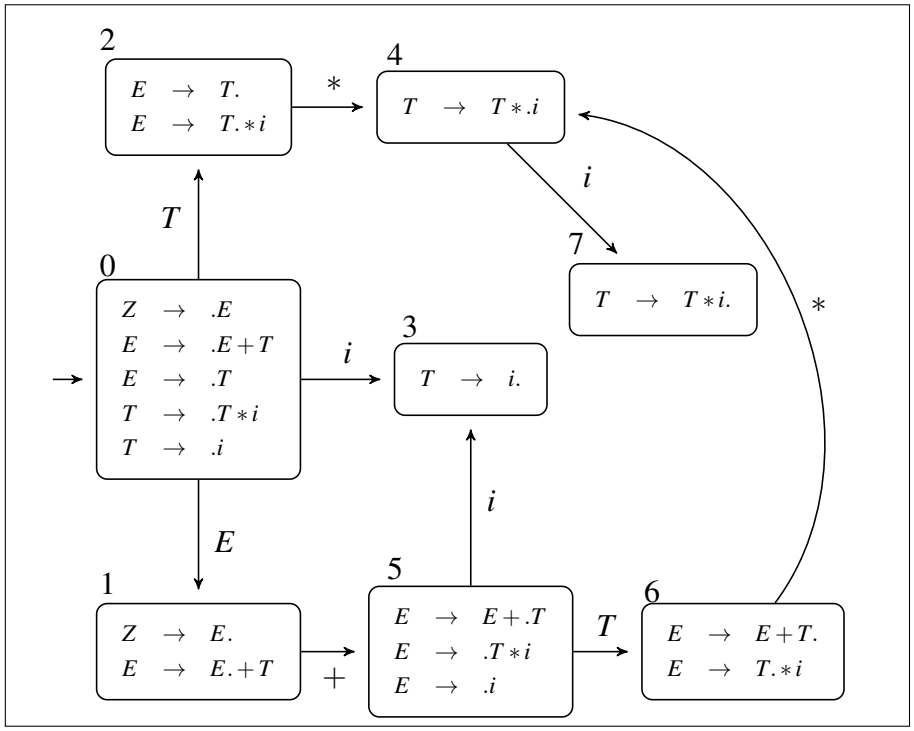


Figure 3.7: Automate LR(0) de G_{13}

La méthode $LR(0)$ repose dans sa décision uniquement sur le contenu de la pile et n'utilise aucun token de prédiction. C'est une méthode faible car peu de grammaires sont $LR(0)$. Par exemple la grammaire G_{13} en haut n'est pas $LR(0)$, il inclut des états ambiguës (2 et 6). Voyons son automate.

3.4.2 Analyse SLR

SLR(for Simple LR) [12] est une amélioration légère sur la méthode précédente dans les cas de conflits décaler/réduire. L'heuristique stipule que devant un état ambiguë à cause d'un conflit décaler/réduire il faut choisir de réduire si le caractère de prédiction fait partie des suivants du non terminal de l'item final, sinon il faut opter pour le décalage. Si malgré cette astuce le conflit persiste, la grammaire est donc non SLR et le recours à d'autre méthodes plus puissante s'impose alors.

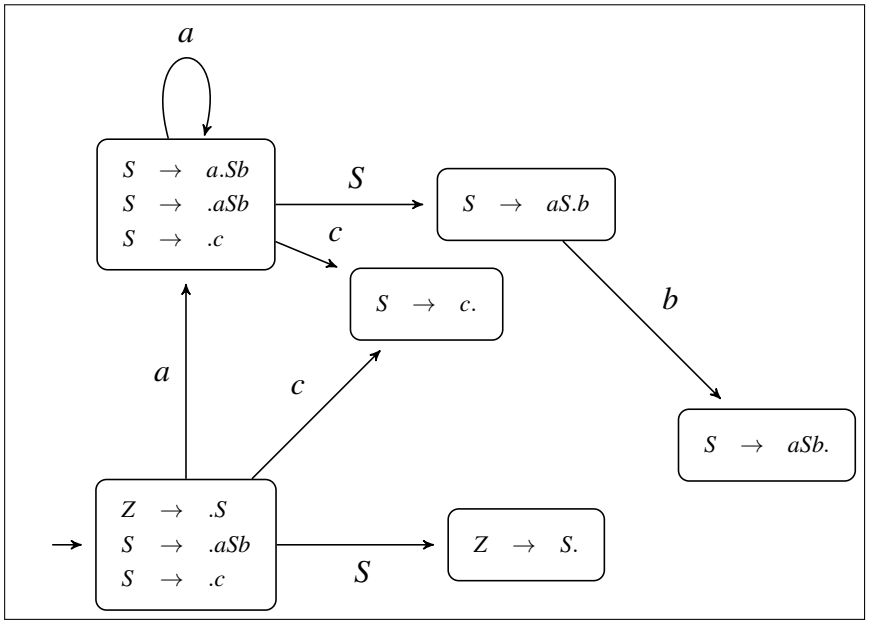


Figure 3.8: Automate LR(0) de G_{14}

Une grammaire non LR(0) mais SLR

Exemple 28. *La grammaire G_{15} présente des conflits décaler/réduire dans deux états.*

$$G_{15} :\rightarrow aSb \mid \varepsilon$$

Comme $Suiv(S) = \{b, \#\}$, l'ambiguïté est levée : On décale à la rencontre d'un a et on réduit à la lecture de b ou # et donc G_{15} est SLR.

Construction de la table d'analyse SLR

Cette table à deux dimensions consigne les actions (décaler, réduire, accepter ou erreur). Elle possède des lignes au nombre des états de l'automate; les symboles de $T \cup N \cup \{\#\}$ représentent quant à eux ses colonnes. Cette table est scindée en deux parties : une partie pour les actions de l'analyseur et la deuxième aux différents transitions de l'automates entre ses états. Ci-après les étapes de sa construction [16].

Pour une grammaire G, si cette construction donne une table mono-définie (chaque case contient au plus une action), G est donc SLR et inversement. La table 3.6 montre la table SLR de la grammaire G_{15} qui remarquons-le est mono-

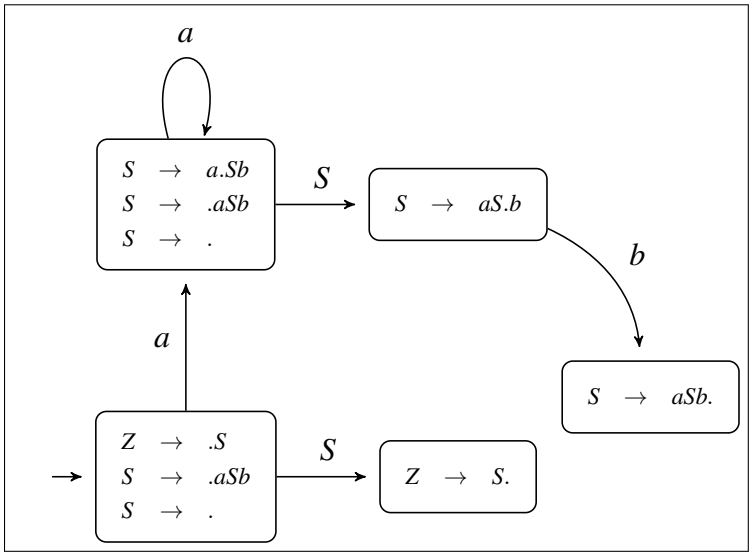


Figure 3.9: Automate LR(0) de G_{15}

Table 3.6: Table SLR de G_{15}

	a	b	#	S
0	D_2			1
1			Accept	
2	D_2			3
3		D_4		
4		$S \to aSb$	$S \to aSb$	

défini‘e.

Une grammaire non SLR

Exemple 29. *La grammaire ci-dessous n'est pas SLR. Son automate comprend un état ambiguë ayant un conflit décaler/réduire. L'analyseur ne sait pas comment agir à la lecture d'un b, car l'état a un item de réduction* $[A \to c.]$ *et un autre de décalage* $[S \to c.b]$*. L'heuristique SLR ne résout pas ce conflit étatnt donnée que les suivants de A inclut b.*

$$G_{16} : \begin{cases} Z \to S \\ S \to A \mid cb \\ A \to aAb \mid c \end{cases}$$

Algorithme 12 Construction de la table d'analyse SLR

- Calculer l'ensemble des items LR(0)
- L'état i est construit à partir de l'ensemble d'items I_i. Ses actions sont :
  1. Si $[A \to \alpha.a\beta] \in I_i$ et $\delta(I_i, a) = I_j$ pour $a \in T$ alors $Tab[i,a] = D_j$
  2. Si $[A \to \alpha.] \in I_i$ pour tout $c \in SUIV(A)$ et $A \neq Z$ alors $Tab[i,c] = R_{A\to\alpha}$
  3. Si $[Z \to S.] \in I_i$ alors $Tab[i,\#] =$ Accepter
- Si $\delta(I_i, A) = I_j$ alors : $Tab[i,A] = j$
- L'état initial est celui sui contient l'item $[Z \to .S]$
- Les entrées vides de la table correspondent à des erreurs.

3.4.3 Analyse LR(1) canonique

Pour doter les méthodes LR d'avantage de puissance, on étend l'ensemble des items LR(0) par l'information de prévision (le caractère lookahead). Un item LR(1) inclut désormais deux composants : L'item de la règle comme dans LR(0) et le symbole de prévision.

Définition 11. *Un item LR(1) est est un objet à deux composants : une production marquée et un caractère de prévision* $[A \to \alpha.\beta, a]$ *où* $a \in T \cup \{\#\}$

Un analyseur LR(1) (automate et table) peut être élaboré de manière similaire à celle de l'analyseur SLR. Il est cependant réputé plus efficace même s'il implique une complexité supérieure. Revenons sur les deux opérations de base **Fermeture** et **Successeur**.

Le principe reste identique, il faut seulement prendre en considération le symbole de prévision. Ci-dessous la procédure pour le calcul des items LR(1).

Voici comment élaborer la table d'analyse LR(1).

Les entrées non définies après application de l'algorithme précédent correspondent à des erreurs. L'état initial I_0 est celui contenant l'item $[Z \to .S, \#]$. Dans ce qui suit nous prenons un exemple d'une grammaire LR(1), nous élaborons l'automate et la table LR(1) et appliquons l'analyse sur une chaîne.

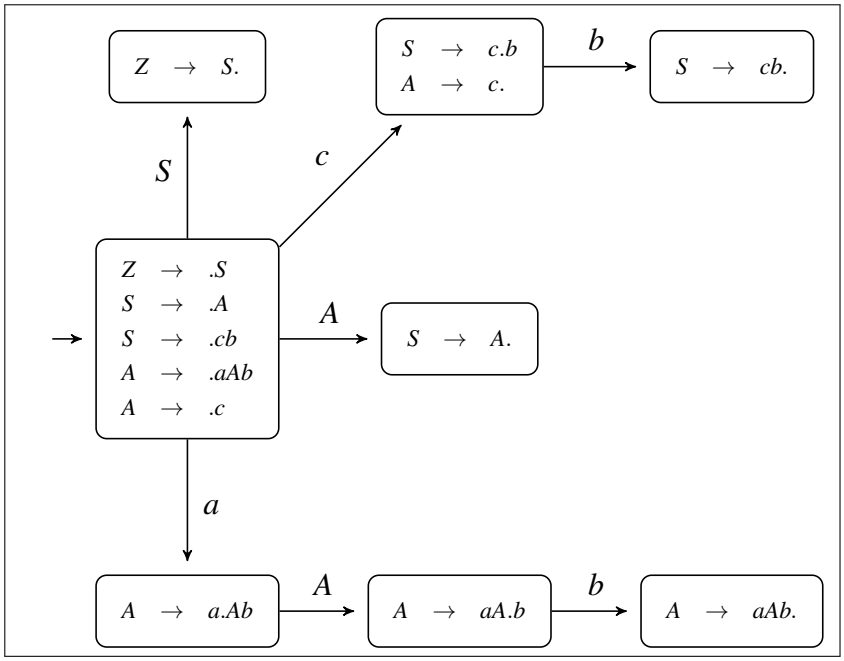


Figure 3.10: Automate LR(0) de G_{16}

Exemple 30. *Soit la grammaire*

$$G_{17} : \begin{cases} Z \to S \\ S \to AA \\ A \to aA \mid b \end{cases}$$

Ci-après la collection des items LR(1) et la table associée :

Pour illustrer le fonctionnement de l'analyseur examinons la chaîne : *abb*.

Table 3.7: Table LR(1) de G_{17}

	a	b	#	S	A
0	D_3	D_4		1	2
1			Accept		
2	D_6	D_7			5
3	D_3	D_4			8
4	$A \to b$	$A \to b$			
5			$S \to AA$		
6	D_6	D_7			9
7			$A \to b$		
8	$A \to aA$	$A \to aA$			
9			$A \to aA$		

Table 3.8: Déroulement de l'analyse LR(1) de G_{17} sur la chaîne *abb*

Pile	Chaîne	Action
#0	*abb*#	Décaler et passer à 3
#0*a*3	*bb*#	Décaler et passer à 4
#0*a*3*b*4	*b*#	Réduire *b* en *A*
#0*a*3*A*	*b*#	Passer à 8
#0*a*3*A*8	*b*#	Réduire *aA* en *A*
#0*A*	*b*#	Passer à 2
#0*A*2	*b*#	Décaler et passer à 7
#0*A*2*b*7	#	Réduire *b* en *A*
#0*A*2*A*	#	Passer à 5
#0*A*2*A*5	#	Réduire *AA* en *S*
#0*S*	#	Passer à 1
#0*S*1	#	Accepter

Algorithme 13 Fermeture d'un ensemble d'items LR(1)

Entrée : I : Un ensemble d'items LR(1)
Sortie : La fermeture $C(I)$

```
Répéter
    Pour tout item [A → α.Bβ, a] ∈ I Faire
        Pour tout B → γ ∈ P Faire
            Pour tout b ∈ DEB(βa) Faire
                I ← I ∪ {[B → .γ, b]}
            Fin Pour
        Fin Pour
    Fin Pour
Jusqu'à Stabilité de C(I)
Retourner C(I)
```

Algorithme 14 Successeur d'un ensemble d'items LR(1)

Entrée : I : Un ensemble d'items LR(1) et un symbole $X \in (T \cup N \cup \{\#\})$
Sortie : La fonction $\delta(I, X)$

```
J ← φ
Pour tout item [A → α.Xβ, a] ∈ I Faire
    J ← J ∪ {[A → αX.β, a]}
Fin Pour
Retourner C(J)
```

3.4.4 Analyse LALR(1)

LALR [14] (pour Look-Ahead LR) est la méthode la plus utilisée en pratique. En effet, la méthode LR(1) bien que puissante, elle implique un calcul prohibitif et requière un espace mémoire très important. LALR à été développée comme un compromis entre simplicité de SLR et puissance de LR(1).

La table d'analyse LALR(1) est obtenue directement à partir des items LR(1) par fusion des états ayants la même partie gauche (cœur). Si cette astuce fournit une table mono-définie, la grammaire est donc LALR. Dans le cas contraire elle est LR(1), mais non LALR(1).

L'application de cette astuce à notre dernier exemple de la grammaire G_{17} permet de réduire les états de 10 à 7. Il aisé de voir que les états 3 et 9, 5 et 7, 4 et 8 peuvent être fusionnés en 3 états seulement. La table d'analyse demeure mono-définie ce qui signifie qu'on peut l'analyser par la méthode LALR.

Algorithme 15 Calcul de la collection des items LR(1)

Entrée : Ǵ : Une grammaire hors contexte augmentée
Sortie : $\mathscr{L}$: La collection des items LR(1)

$\mathscr{L} \leftarrow \{[Z \rightarrow .S, \#]\}$
Répéter
  Pour tout item $I \in \mathscr{L}$ **Faire**
    Pour tout $X \in T \cup N \cup \{\#\}$ **Faire**
      Si $\delta(I,X) \neq \emptyset \wedge \notin \mathscr{L}$ **Alors**
        $\mathscr{L} \leftarrow \mathscr{L} \cup \delta(I,X)$
      Fin Si
    Fin Pour
  Fin Pour
Jusqu'à Stabilité de $\mathscr{L}$

Algorithme 16 Construction de la Table LR(1)

Entrée : La collection des items LR(1)
Sortie : La table LR(1) renseignée

Si $[A \rightarrow \alpha.a\beta, b] \in I_i \wedge \delta(I_i, a) = I_j$ **Alors**
  $Tab[i,a] \leftarrow D_j$
Fin Si
Si $[A \rightarrow \alpha., a] \in I_i$ **Alors**
  $Tab[i,a] \leftarrow R_{A \rightarrow \alpha}$ Avec $A \neq Z$
Fin Si
Si $[Z \rightarrow S., \#] \in I_i$ **Alors**
  $Tab[i,\#] \leftarrow$ Accept
Fin Si
Si $\delta(I_i, A) = I_j$ **Alors**
  $Tab[i,A] \leftarrow j$
Fin Si

3.5 Générateurs d'analyseurs syntaxiques avec Bison

À l'instar des générateurs de scanners, une panoplie de générateurs d'analyseurs syntaxiques sont développés en utilisant différents langages de programmation et plateformes. Ces outils reçoivent la spécification syntaxique du langage à implémenter et produisent le code du parser correspondant. Certains de ces outils adoptent l'approche descendante (JavaCC [33], Coco/R [15], etc.), d'autre reposent sur la méthode LALR et sont conforment de ce fait à l'approche ascendante (Bison [21], PLY [1], CUP [7], etc.). Dans ce qui suit nous présentons l'essentiel pour exploiter Bison afin de générer un analyseur syntaxique.

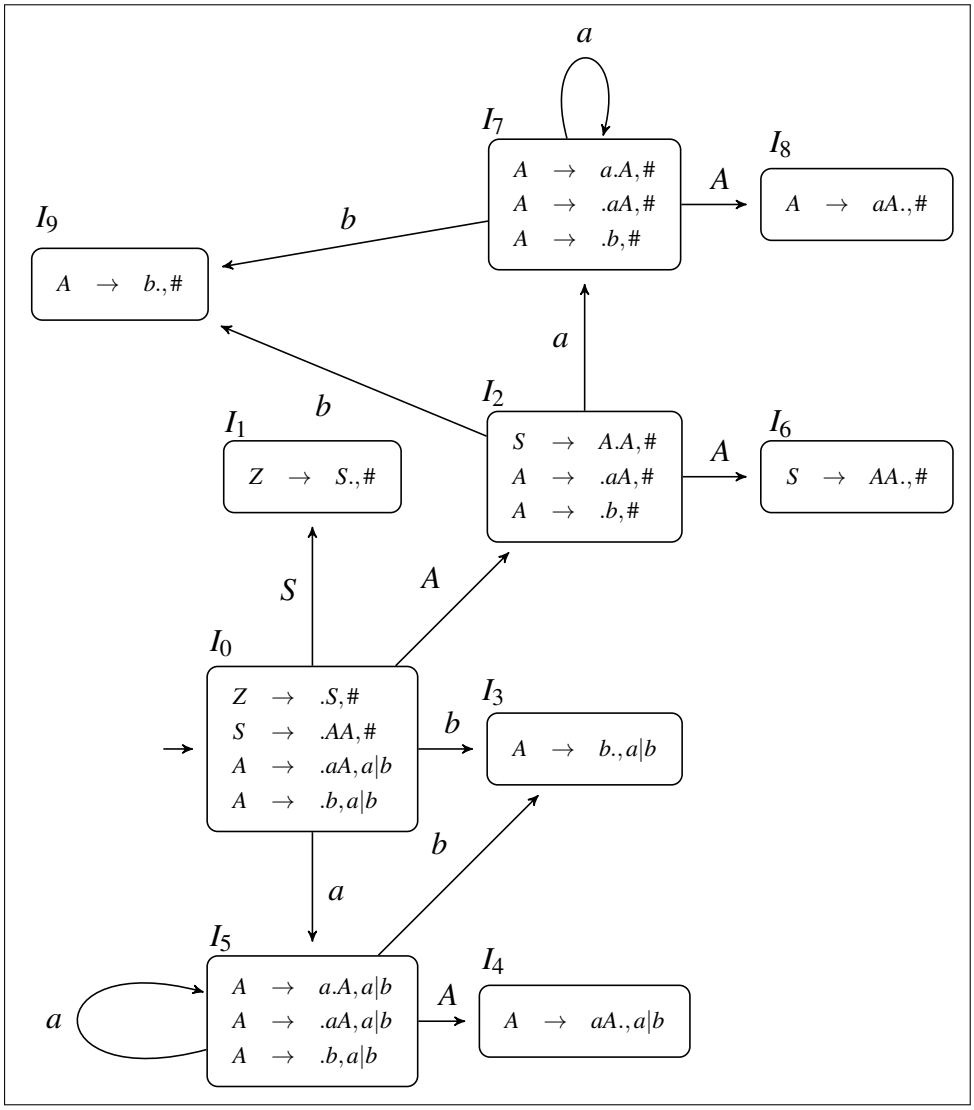


Figure 3.11: Automate LR(1) de G_{17}

Bison [21] est une version améliorer et libre du fameux générateur Yacc d'Unix [17]. C'est un générateur d'analyseurs syntaxiques qui reçoit la spécification syntaxique et produit le code source de l'analyseur en langage C. L'entrée de Bison est la spécification syntaxique du langage c-à-d une grammaire à contexte libre LALR en format BNF facile à lire par un programme. Bison compile les règles et les déclarations dans le fichier d'entrée afin de produire la table d'analyse LALR codée en langage C sous la forme d'instructions switch dans la routine **yyparse()**.

Similairement à Flex, un fichier (d'extension .y par convention) d'entrée à Bison inclut, comme illustre la Figure 3.12, quatre parties : une pour les déclarations

globales, une deuxième pour les déclarations propres à Bison, la partie règles de production de la grammaire et en fin un code suplémentaire.

```
%{
  Declaration C
%}
  Declaration Bison
%%
  Regles de la grammaire
%%
  Code C additionnel
```

Figure 3.12: Structure d'un fichier de spécification Bison

3.5.1 Partie déclaration C

Cette partie facultative et délimitée par la couple %{ et %} engloble les macros, les includes, les définitions et les déclarations des variables et fonctions utilisées dans la parties actions des règles de la grammaire.

3.5.2 Partie déclaration Bison

Ici on définit les symboles de la grammaire (terminaux et non terminaux), les priorités et précédences des opérateurs et les types de données pour les actions. Ci-dessous sont résumé les directives de Bison.

3.5.3 Partie règles de la grammaire

Cette section contient exclusivement les règles de la grammaire sous la forme : *A* : *MDP* où *A* est un non terminal. Si la règles possède de multiple alternatives, elles seront séparées par une bare verticale. L'ensemble des règle est terminé par un point virgule. Voici un exemple :

```
exp  :  exp PLUS  exp
     |   exp  MOINS  exp
     |   exp  FOIS  exp
     ;
```

Pour assurer une sémantique au langage analysé, des actions sémantiques écrites en langage C peuvent être insérées après chaque règle de production. Un exemple serait d'additionner les valeurs associées aux symboles pour la première règle.

```
exp  :  exp PLUS  exp        {$$ = $1 + $3;}
```

3.5.4 Partie code additionnel

Cette dernière est optionnelle comme la première et seront copiées (une à la fin et l'autre au début) telle quelles dans le code source de l'analyseur. On peut y ajouter tout ce que nous voulons accomplir en plus du code du parser proprement dit.

Voici ci-après un résumé des principales directives de Bison.

- Les unités lexicales seront déclarées par la directive **%token nom_de_l'unité**
- Les directives **%left (resp. %right)** permettent de définir une règle d'associativité gauche (resp. droite), par exemple %left + : signifie que l'expression a+b+c sera évaluée : (a+b)+c. Noter qu'un token déclaré par %left (resp. %right) n'a pas besoin d'être introduit par une directive %token. **%nonassoc** : pour les opérateurs non associatifs, et **%start** pour préciser l'axiome, sinon le premier dans la partie règles sera considéré comme tel.
- L'ordre de précédence est déterminé par l'ordre d'apparition des tokens (les premiers listés ont la priorité la plus faible). Si plusieurs tokens possèdent le même ordre de précédence, il seront déclarés en même temps sur la même ligne.
- On introduit la grammaire dans la partie délimitée par %%. Les productions d'un non terminal seront séparées sur différentes lignes par | comme dans la notation BNF. Une ligne vide remplace l'alternative vide (ε)
- Si une règle de production a été reconnue l'action associée sera exécutée. Comme en Flex, une action est un bloc de code en C. les $\$_n$ désignent les numéros des symboles comme ils apparaissent dans le MDP et **$$** fait référence au MGP. Exemple : $E \longrightarrow E + E \ \{\$\$ = \$_1 + \$_3;\}$.
- La fonction **main()** de l'outil appelle **yyparse()** pour traiter l'entrée, qui appelle à son tour la fonction **yylex()** afin de récupérer les tokens de l'analyseur

lexical. Si une erreur est rencontrée, l'analyseur appelle **yyerror(m)**, où **m** est le message d'erreur devant être renvoyé.

- La déclaration **%union** définie la collection des types de données utilisés dans les actions de l'analyseur. Par exemple la déclaration suivante indique que deux types (un réel et une chaîne seront exploités).

```
\%union {
  double val;
  char* nom;
}
```

  Ces types peuvent être utilisés avec les directives %token et %type afin d'indiquer les types pour les symboles de la grammaire.

Voici comme résumé un exemple complet d'un code Bison pour la grammire des expressions arithmétiques :

3.6 Gestion des erreurs

Différentes sortes d'erreurs peuvent être commises dans les codes sources. Celles-ci peuvent s'intercaler dans les différentes phases d'analyse. Quelque soit l'erreur, le compilateur doit la prendre en charge efficacement. Ainsi la gestion des erreurs inclut :

- La détection rapide et précise de l'erreur (spécifier sa nature et indiquer la position exacte dans le texte source)
- Le traitement efficace de l'erreur de façon à ne pas compromettre le processus d'analyse

Les erreurs communes se répartissent en quatre types :

- **Lexicales** comme la malformation des unités lexicales (mots-clés, identificateurs, etc.) qui résultent souvent des coquilles de frappes
- **Syntaxiques** : le non respect de la syntaxe du langage, comme les séparateurs déplacés ou oubliés, les parenthèses non équilibrées; ...
- **Sémantiques** qui n'observent pas à titre d'exemple le systèmes de types, les déclarations manquantes, etc.

```
%{  #include <stdio.h>
  #include<string.h>
  #include "calc.h"
  int yylex(void);
  void yyerror(char *);
%}
%union {  double fl;
      char car; }
%type <fl> expr exp condition
%token <fl> NB
%token <car> var egal diff  et ou si alors
%token '<'
%token '>'
%token sortie
%token '(' ')'
%left '+'
%left '*'
%right '='
%nonassoc na
%%
prg:    /* rien (vide) */
expr';' '\n'      { printf("%f\n> ", $1); }
  | si '(' condition ')' alors expr {if ($3) { $1;}}
  | var '=' expr    { inserer(symbole, $1 , $3);}
  | sortie      { printf("> Au revoir !");exit(0);}
  ;
expr      : NB        { $$ = $1 ; }
  | var       { $$ = recuperer(symbole, $1);}
  | expr '+' expr   { $$ = $1 + $3; }
  | expr '*' expr   { $$ = $1 * $3; }
  | '-' expr %prec na   { $$ = - $2; }
  | '(' expr ')'    { $$ = $2; }
  ;
condition : exp ou exp      { $$ = $1 && $3;}
  | exp et exp      { $$ = $1 && $3;}
  ;
exp   : expr '<' expr   { $$ = $1 < $3; }
  | expr '>' expr   { $$ = $1 > $3; }
  | expr "==" expr    { $$ = $1 == $3; }
  | expr "!=" expr    { $$ = $1 != $3; }
  ;
%%
void yyerror(char *s) {
  fprintf(stderr, "%s\n> ", s); }
void inserer (struct table **t, char c, double val){
  /* inserer une variable dans une llc */
  while (t != NULL){
    ...
  }
double recuperer (struct table **t, char c){
  /* rechercher une variable  et recuperer sa valeur*/
  ....
return val;
}
int main(void) {
  printf("> "); *symbole = NULL; yyparse(); return 0;
}
```

Figure 3.13: Code Bison complet

- **Logiques** représentant les erreurs de raisonnement de la part du programmeur comme les boucles infinies

Généralement, les deux premières classes d'erreurs et une partie de la troisième peuvent être détectées sans failles. Cela est dû aux outils des analyseurs (automates, tables, préfixes viables, etc.) qui offrent une détection quasi-instantanée. Cependant, beaucoup d'erreurs sémantiques et la totalité des erreurs logiques sont loin d'être maîtrisées par les compilateurs en raison de leur nature extrêmement complexe.

Plusieurs approches ont été adoptées pour traiter les erreurs. Le principe est surtout la correction et l'efficacité. Les plus connues des techniques de récupération suite à une erreur sont : le mode panique, le mode au niveau du syntagme et la production des erreurs [16].

3.6.1 Récupération en mode panique

Cette technique est la plus simple des stratégique. À la rencontre d'une erreur, le compilateur supprime généralement plusieurs tokens jusqu'à la détection d'un qui appartient à un ensemble dit de synchronisation ce qui permet de reprendre l'analyse promptement. Les symboles de synchronisation englobent par exemple les séparateurs commes les point-virgules ou les parenthèses, les délimiteurs de blocs, etc.

3.6.2 Récupération au niveau du syntagme

À l'opposé de la précédente stratégie, ici le compilateur n'élimine pas des entités mais plutôt propose des corrections locales du genre remplacer un , par un ; inverser fi à if, etc. Bien entendu, les rectifications introduites ne doivent pas générer d'autres situations ingérables ou faire coincer l'analyse dans une boucle infinie par exemple.

3.6.3 Production d'erreurs

Ici l'approche est différente. La grammaires est augmentée dès la conception par des constructions qui gèrent les erreurs prévisibles. Cette manière d'anticiper les erreurs susceptible de surgir offre un traitement plus lisible et compréhensible à l'utilisateur

Exercices du chapitre 3

ANALYSE SYNTAXIQUE

Exercice 1

Quel est le nombre de mots et d'arbres syntaxiques distincts générés par la grammaire ci-contre :

$$G : \begin{cases} S & \to A1 \mid 1B \\ A & \to 10 \mid C \mid \varepsilon \\ B & \to C1 \mid \varepsilon \\ C & \to 0 \mid 1 \end{cases}$$

Exercice 2

Ci-après une grammaire des expressions lisp simplifiées :

$$G_1 : \begin{cases} lexp & \to atom \mid list \\ atom & \to number \mid identifier \\ list & \to (lexpseq) \\ lexpseq & \to lexpseq \quad lexp \mid lexp \end{cases}$$

(a) Écrire la dérivation la plus à gauche/à droite de la chaîne : $(a \quad 23 \quad (m \quad x \quad y))$

(b) Donner son arbre syntaxique

Exercice 3

Éliminer la récursivité gauche dans les grammaires suivantes :

$$G_1 : \begin{cases} S & \to S\backslash S \mid SS \mid A \\ A & \to S* \mid (S) \mid a \mid b \end{cases} \qquad G_2 : \begin{cases} S & \to AA \mid 0 \\ A & \to SS \mid 1 \end{cases}$$

$$G_3 : \begin{cases} S & \to AS \mid b \\ A & \to SA \mid a \end{cases} \qquad G_4 : \begin{cases} A & \to Cd \\ B & \to Ce \\ C & \to A \mid B \mid f \end{cases}$$

Exercice 4

Soit la grammaire : $G_1 : S \to 0S1S \mid 2S \mid \varepsilon$

(a) Écrire un analyseur syntaxique pour G_1 en utilisant la descente récursive.

(b) Analyser la chaîne : 012

Exercice 5

Soit la grammaire G_1 ci-contre.

$$G_1 : \begin{cases} S & \to aB \mid bA \mid c \\ A & \to aS \mid bA \\ B & \to b \end{cases}$$

(a) Montrer que G_1 est LL(1)

(b) Écrire un analyseur syntaxique pour G_1 en utilisant la descente récursive.

(c) Analyser la chaîne : *babbac*

Exercice 6

Soit la grammaire : $G_1 : \begin{cases} S \to a \mid b \mid (T) \\ T \to T, S \mid S \end{cases}$

(a) Éliminer la récursivité à gauche. Montrer que la grammaire transformée est LL

(b) Donner la table d'analyse LL, et analyser la chaîne : $(a,(b,a),a)$

Exercice 7

Les grammaires suivantes sont-elles LL(k) ?

$$G_1 = \begin{cases} S & \to AB \mid aSb \mid CSB \\ A & \to bA \mid \varepsilon \\ B & \to dB \mid \varepsilon \\ C & \to cC \mid e \end{cases} \qquad G_2 = \begin{cases} S & \to ASB \mid \varepsilon \\ A & \to aAc \mid c \\ B & \to bBA \mid \varepsilon \end{cases} \qquad G_3 = \begin{cases} S & \to A \mid B \mid a \\ A & \to bBA \\ B & \to dB \mid \varepsilon \end{cases}$$

Exercice 8

On s'intéresse à la construction d'un analyseur syntaxique pour la grammaire G ci-après, où $\mathbb{T} = \{., id, @\}$, $\mathbb{N} = \{S, N\}$: $G : \begin{cases} S & \to N@N.\mathbf{id} \\ N & \to \mathbf{id} \mid \mathbf{id}.N \end{cases}$

(a) Donner l'arbre de dérivation pour la chaîne **id@id.id.id**

(b) Peut-on analyser cette grammaire par une méthode descendante ? Pourquoi ? Si votre réponse est négative, la transformer donc.

(c) La grammaire transformée obtenue dans la question précédente est-elle LL(1) ? Justifier.

(d) En étudiant le langage généré par G, donner une grammaire équivalente qui soit LL(1), puis tracer sa table d'analyse LL.

(e) Analyser la chaîne introduite en (a).

Exercice 9

Soit la grammaire G ci-après, où $\mathbb{T} = \{a, b\}$, $\mathbb{N} = \{S, N\}$:

$$G_2 : \begin{cases} S & \to abN \mid \varepsilon \\ N & \to Saa \mid b \end{cases}$$

(a) Calculer les ensembles DEB_1 et SUIV_1 de S et N

(b) Montrer que G_2 n'est pas LL(1).
On étend les ensembles DEB et SUIV à k caractères $k > 1$ comme suit :

$$DEB_k(X) = \{w \in \Sigma^* \mid X \to^* w\alpha \text{ et } |w| = k, \text{ ou } X \to^* w \text{ et } |w| < k\}$$

$$SUIV_k(X) = \{w \in \Sigma^* \mid S \to^* \alpha X \beta \text{ et } |w| \in DEB_k(\beta)\}$$

(c) Calculer les ensembles DEB_2 et SUIV_2 de S et N.
Sachant que la construction d'une table LL(k) et la même que pour une analyse LL(1), en remplaçant le terminal de prévision par DEB_k (ou $SUIV_k$) terminaux.

(d) Tracer la table LL(2) de G_2, puis analyser la chaîne : *ababbaa*.

Exercice 10

Étant donné la grammaire G ci-contre : $S \to S\bigstar S \mid \blacktriangle$
Discuter en justifiant si G est :

(a) Ambiguë (b) LL (c) SLR (d) LALR

(e) Simuler l'exécution de l'analyseur approprié de G sur la chaîne : ▲★▲★▲

Exercice 11

On s'intéresse à la construction d'un analyseur syntaxique pour la grammaire G ci-contre :

$$G : \begin{cases} S & \rightarrow aAS \mid bA \\ A & \rightarrow cA \mid d \end{cases}$$

(a) Trouver les ensembles DEB et SUIV des non terminaux de G

(b) Calculer la collection des items LR[0]. G est-elle SLR(1) ? Justifier.

(c) Construire la table d'analyse SLR(1) pour G

(d) Simuler votre analyseur sur la chaîne : **acdbd**

Exercice 12

Soit la grammaire G ci-contre :

$$G : \begin{cases} E & \rightarrow (L) \mid a \\ A & \rightarrow EL \mid E \end{cases}$$

(a) Construire l'automate des items LR(0) et la table d'analyse SLR, puis analyser ((a)a(a a)).

(b) Construire l'automate des items LALR(1) puis la table correspondante en propageant les caractères de prévision via l'automate des items LR(0)

Exercice 13

Montrer que la grammaire suivante est LR(1) mais non LALR(1) :

$$G : \begin{cases} S & \rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A & \rightarrow c \\ B & \rightarrow c \end{cases}$$

Exercice 14

Soit la grammaire G suivante :

$$G : \begin{cases} S & \rightarrow AS \mid b \\ A & \rightarrow SA \mid a \end{cases}$$

(a) G est-elle SLR. Si oui, construire sa table d'analyse.

(b) Cette grammaire est elle LR ? LALR ?

Travaux pratiques

Mini projet 1 de Compilation

1 Objectif

Le but de ce TP est de réaliser la première phase de la compilation (analyse lexicale) pour un mini-langage de programmation de deux manières différentes.

2 Spécification lexicale

Nous avons défini le mini-langage de programmation **jad** composé des éléments suivants :

- Les mots clés (**KWD**) insensibles à la casse : **si, sino, is, tq, qt, rpt, jsq, lre, ecr, vrai, faux**
- Les entiers naturels (**INT**) non signés composés par des suites non vides de chiffres
- Les identificateurs (**ID**) : les suites de lettres ou de chiffres commençant par une lettre
- Les opérateurs (**OPR**) : $+ \quad - \quad * \quad / \quad = \quad < \quad <= \quad > \quad >= \quad == \quad != \quad \&\& \quad || \quad !$
- Séparateurs (**SP**) : ; , ( ) :
- Les blancs (**BLC**): les suites non vides de : espace, tabulation et fin de ligne

3 Travail à faire

1. Écrire un analyseur lexical ad-hoc (codé en dur) pour **jad** Vous êtes libre dans le choix du langage de réalisation (C, Java, Python, etc.). L'analyseur lexical doit recevoir le code source dans un fichier texte et fournir son résultat, après avoir retourner le token rencontré, dans un autre fichier de sortie en respectant le format suivant :

Classe	Ligne	Colonne	Lexème
KWD	1	4	si
INT	2	5	123
ID	3	1	somme1
...	...	...	...

   Nous enrichissons maintenant notre mini-langage par les nouveaux tokens ci-après :

   (a) Les flottants (**FLT**) : les nombre réels signés en notation scientifique

(b) Les chaînes de caractères (**CHN**) : elles sont incluses entre double quotes "...". Dans une chaîne une séquence '\c' dénote le caractère c avec les exceptions suivantes :

- \b backspace
- \t tab
- \n newline
- \f formfeed

Vous devez respecter les restrictions suivantes sur les chaînes :

- Une chaîne ne peut contenir le nul (le caractère \0) ni le EOF
- Une chaîne ne peut dépasser les limites des fichiers
- Une chaîne ne peut inclure de nouvelles lignes non escapées. Par exemple :
  "Cette chaîne \
  est correcte "
  par contre la suivante est invalide :
  "Cette chaîne
  est incorrecte "

(c) les commentaires (**CMT**) au style du C : sur une seule ligne introduits par "//" ou ceux multi-lignes encadrés par les couples "/*" et "*/". Cette dernière forme peut être imbriquée.

2. Réécrire le même analyseur lexical en utilisant cette-fois l'outil **flex** (même entrée/sortie en fichiers textes).

4 Gestion des erreurs

Toutes les erreurs doivent être retournées au analyseur syntaxique. Elles seront communiquées en retournant un token spécial **ERROR** et suivant les exigences ci-après :

- Si un caractère invalide est rencontré, la chaîne contenant ce caractère doit être retournée comme chaîne d'erreur,
- Si une chaîne inclut un newline non escapé reporter l'erreur : "Constante chaîne de caractères non terminée", et reprendre l'analyse au début de la ligne suivante,
- Dans le cas d'une chaîne trop longue reporter "Constante chaîne de caractères trop longue" dans la chaîne erreur du token **ERROR**. Si la chaîne contient un caractère invalide (le nul par exemple) reporter l'erreur "Chaîne contenant le caractère nul", dans les deux cas l'analyse reprendra après la fin de la chaîne. La fin d'une chaîne est définie soit comme :
  1. le début de la prochaine ligne si un newline non escapé est rencontré après ces erreurs ou
  2. après les deux quottes de fermeture "
- Si un commentaire reste ouvert alors qu'un EOF est rencontré reporter l'erreur :" Fin de fichier dans un commentaire" ne pas considérer alors ce token étant donné que la terminaison est manquante. De même, si EOF est rencontrés avant la quotte de fermeture d'une chaîne reporter "Fin de fichier dans une constante chaîne"
- Si vous rencontrez "*/" en dehors d'un commentaire reporter "*/ sans correspondance" au lieu de la tokeniser en * et /.

5 Construction de l'analyseur

1. Pour les codages en dur utiliser votre IDE favori
2. Compiler la spécification Flex par : **Flex -o scanner.c scanner.l** : produira le fichier **scanner.c** à partir de scanner.l
3. Compiler le tout par : **gcc -o analyseur scanner.c -lfl** : produira l'exécutable **analyseur** à partir de l'ensemble des fichiers
4. Pour le codage des tokens vous pouvez soit les définir comme de simples entiers, ou utiliser l'outil **Bison**. Dans ce dernier, on définit un token en le précédant de l'option **%token**. Compiler le fichier bison en utilisant l'option **-d** pour produire la définition de vos tokens en fichier entête **.h** à inclure dans votre fichier flex.
5. Invoquer votre analyseur par : **./analyseur** : où vous pouvez taper votre code source en **jad** ou en le passant à travers un fichier texte et invoquer l'analyseur par la commande **./analyseur** $<$ **votre-fichier-source**

Mini projet 2 : Expressions arithmétiques étendues

1 Objectif

Après avoir exploré les différentes techniques d'analyse syntaxique. On vous propose de réaliser un petit analyseur syntaxique pour les expressions arithmétiques en utilisant les outils **Bison et Flex**. le point de démarrage sera deux fichiers de spécification lexicale(Flex) et syntaxique(Bison) à enrichir graduellement par des constructions simples.

2 Bison

Bison est la version gnu (libre) du fameux générateur automatique d'analyseurs syntaxique yacc. Ce dernier crée un analyseur **LALR(1)** extensible par des actions sémantiques et des règles de désambiguïsation pour les grammaires non LALR(1). Le format du fichier de spécification est semblable à celui de Flex à savoir : une partie déclaration, une deuxième de définition de règles et une dernière pour le code supplémentaire.

2.1 Spécification Bison

(a) Les unités lexicales seront déclarées par la directive **%token**, qui donne lieu à l'insertion d'une déclaration par **#define** dans le fichier entête calc.tab.c produit par Bison.

(b) Les directives **%left (resp. %right)** permettent de définir une règle d'associativité gauche (resp. droite), par exemple %left + : signifie que l'expression a+b+c sera évaluée : (a+b)+c. Noter qu'un token déclaré par %left (resp. %right) n'a pas besoin d'être introduit par une directive %token. **%nonassoc** : pour les opérateurs non associatifs.

(c) L'ordre de précédence est déterminé par l'ordre d'apparition des tokens(les premiers listés ont la priorité la plus faible). Si plusieurs tokens possèdent le même ordre de précédence, il seront déclarés en même temps sur la même ligne.

(d) On introduit la grammaire avec les actions sémantiques associées aux différentes productions. Ces dernières seront séparées sur différentes lignes par 'l' comme dans la notation BNF. Une ligne vide remplace l'alternative vide (ε)

(e) Si une règle de production a été reconnue l'action associée sera exécutée. Comme en Flex, une action est un bloc de code en C. les $\$_n$ désignent les numéros des symboles comme ils apparaissent dans le MDP et **$$** fait référence au MGP. Exemple : $E \longrightarrow E + E \ \{\$\$ = \$_1 + \$_3;\}$

(f) La fonction **main()** de l'outil appelle **yyparse()** pour traiter l'entrée, qui appelle à son tour la fonction **yylex()** afin de récupérer les tokens. Si une erreur est rencontrée, l'analyseur appelle **yyerror(m)**, où m est le message d'erreur devant être renvoyé.

(g) Bison maintient **2 piles séparées** : une première pour l'analyse syntaxique qui contient les symboles de la grammaire et une deuxième de valeurs contenant les attributs de ces symboles.

(h) La communication entre Bison et Flex est assurée via la variable **yyval**, son type est le même que celui des éléments de la pile des valeurs

2.2 Construction de l'analyseur

1. Compiler la spécification Flex par : **Flex -o calc.c calc.l** : produira le fichier calc.c à partir de calc.l
2. Compiler la spécification Bison par : **Bison -d calc.y** : produira calc.tab.h et calc.tab.c à partir de calc.y
3. Lier le tout par : **gcc calc.c calc.tab.y -o calculateur** : produira calculateur à partir de l'ensemble des fichiers .c et.h
4. Invoquer votre analyseur par : **./calculateur** : où vous pouvez taper vos expressions.

2.3 Travail demandé

(a)
1. A partir du groupe de travail télécharger les deux fichiers de spécification calc.l et calc.y (dossier : 3I/compilation/TP)
2. Compiler et tester votre mini-calculette sur des exemples d'expressions simples avec les opérateurs + et -
3. Compléter la grammaire par l'introduction des opérateurs binaires : *, /
4. Remarquer les messages des conflits détectés par Bison, puis introduire des règles d'associativité/précédence et réexaminer le résultat.
5. Ajouter l'opérateur unaire -
6. Ajouter à votre grammaire la possibilité d'introduction de variable dont le nom est composé d'une seule lettre (autoriser toutes les lettres de l'alphabet sauf le Q : en majuscule !), ainsi que l'instruction d'affectation simple (associative à droite). Exemple : $x = 14$
7. Ajouter les expressions parenthésées, du genre ((1+2)*x/(1-3))/12
8. Ajouter une production pour terminer les calcul et quitter la calculette si on tape la lettre Q (en majuscule)
9. Modifier la grammaire pour que toute expression soit terminée par un point-virgule suivie d'une nouvelle ligne
10. Le type des attribut étant ENTIER par défaut modifier-le pour permettre des expressions avec des flottants (le type double en C)

(b) S'il vous reste du temps

1. Permettre cette fois-ci des noms de variables quelconques dont la longueur n'excède pas 5 caractères.
2. Ajouter la structure conditionnelle simple : **Si (condition) Alors Action.**

3 Exemples

```
> 1+1
> 2
> (2+5)
> 7
> −6−3
> −9
> x=y=12
> 12
> x+y
> 24
> x+3∗(y/2−1)
> 27
> 1+2−
> syntax error
> 1+2:5
> caractere invalide syntaxe error
> Q
> Au revoir !
```

Mini projet 3 : Parser pour jad

1 Objectif

Le but de ce TP est de réaliser l'analyseur syntaxique pour le mini-langage de programmation **jad** de deux manières différentes.

2 Specification lexicale de jad (rappel)

jad est compose des token suivants :

- Les mots clés (**KWD**) insensibles à la casse : **si, sinon, is, tq, qt, rpt, jsq, lre, ecr, vrai, faux**
- Les entiers naturels (**INT**) non signés composés par des suites non vides de chiffres
- Les identificateurs (**ID**) : les suites de lettres ou de chiffres commençant par une lettre
- Les opérateurs (**OPR**) : $+$ $-$ $*$ $/$ $<$ $<=$ $>$ $>=$ $==$ $!=$ $\&\&$ $||$ $!$
- Séparateurs (**SP**) : ; , ( ) :
- Les commentaires (**CMT**) au style du C : sur une seule ligne introduits par "//" ou ceux multi-lignes encadrés par les couples "/*" et "*/". Cette dernière forme peut être imbriquée.
- Les blancs (**BLC**): les suites non vides de : espace, tabulation et fin de ligne

3 Spécification syntaxique

La grammaire du langage **jad** est donnée ci-après : les symboles en gras sont des mots clés ou des terminaux, les non terminaux sont entre les couples $< et >$

$$
\begin{aligned}
< prgm > &\rightarrow < decl >< bloc > \\
< decl > &\rightarrow < decl >< type >< list > \textbf{;} \mid \varepsilon \\
< list > &\rightarrow < list > \textbf{,} < id > \mid < id > \\
< type > &\rightarrow \textbf{int} \mid \textbf{flt} \mid \textbf{bol} \\
< bloc > &\rightarrow < bloc > \textbf{;} < instr > \mid < instr > \\
< instr > &\rightarrow < i-aff > \mid < i-si > \mid < i-tq > \mid < i-rpt > \mid < i-lre > \mid < i-ecr > \\
< i-aff > &\rightarrow < id > \textbf{:=} < exp > \\
< i-si > &\rightarrow \textbf{si} < exp > \textbf{:} < bloc > \textbf{is} \\
< i-tq > &\rightarrow \textbf{tq} < exp > \textbf{:} < bloc > \textbf{qt} \\
< i-rpt > &\rightarrow \textbf{rpt} < bloc > \textbf{jsq} < exp > \\
< i-lre > &\rightarrow \textbf{lre} < id > \\
< i-ecr > &\rightarrow \textbf{ecr} < exp > \\
< exp > &\rightarrow < exp >< op >< term > \mid < term > \\
< term > &\rightarrow < term >< opb >< fact > \mid < fact > \\
< fact > &\rightarrow < nb > \mid < id > \mid \textbf{(} < exp > \textbf{)} \mid < opu >< exp > \mid \textbf{vrai} \mid \textbf{faux} \\
< op > &\rightarrow < op1 > \mid < op2 > \\
< op1 > &\rightarrow \textbf{+} \mid \textbf{-} \mid || \\
< op2 > &\rightarrow < \mid <= \mid > \mid >= \mid \textbf{==} \mid != \\
< opb > &\rightarrow \textbf{*} \mid \textbf{/} \\
< opu > &\rightarrow \textbf{!} \mid \textbf{-} \\
< nb > &\rightarrow < chiff >< nb > \mid < chiffre > \\
< id > &\rightarrow < car >< suite > \mid < car > \\
< suite > &\rightarrow < chiff >< suite > \mid < car >< suite > \mid \varepsilon \\
< chiff > &\rightarrow \textbf{0} \mid \textbf{1} \mid ... \mid \textbf{9} \\
< car > &\rightarrow \textbf{a} \mid \textbf{b} \mid ... \mid \textbf{z} \mid \textbf{A} \mid \textbf{B} \mid ... \mid \textbf{Z}
\end{aligned}
$$

4 Travail à faire

1. Écrire l'analyseur syntaxique pour ce mini-langage par la descente récursive (Vous serez amener à transformer certaines instructions !). Pour cette partie utiliser le langage de votre choix.

2. Réécrire le même analyseur syntaxique en utilisant l'outil **Bison**. Dans ce cas, vous pouvez ne pas altérer votre grammaire (Bison est un analyseur ascendant).

5 Construction des l'analyseurs

1. Compiler la spécification Flex par : **Flex -o scanner.c scanner.l** : produira le fichier **scanner.c** à partir de scanner.l

2. Compiler la spécification Bison par : **Bison -d parser.y** : cette commande produira le fichier **parser.tab.h** qui contient le codage des tokens qui seront transmis par l'analyseur lexical et le fichier **parser.tab.c** qui contient le code source de l'analyseur syntaxique à partir de la spécification contenue dans le fichier parser.y

3. Compiler le tout par : **gcc -o analyseur scanner.c parser.tab.c -lfl** : produira l'exécutable **analyseur** à partir de l'ensemble des fichiers .c et.h

4. Invoquer votre analyseur par : **./analyseur** : où vous pouvez taper vos codes sources en **jad** Vous pouvez également introduire votre code dans un fichier texte et invoquer l'analyseur par la commande **./analyseur** $<$ **votre-fichier-source**

Bibliography

[1] (2020). Beazley, d. ply (python lex-yacc). https://www.dabeaz.com/ply/. (Cité en pages 41 et 79.)

[2] (2020). Klein, g., rowe, s., and décamps, r., jflex : https://jflex.de/. (Cité en page 41.)

[3] (2020). Paxson, v., estes, w., and millaway, j. flex : fast lexer ; main page : https://www.gnu.org/software/flex/. (Cité en page 41.)

[4] Aho, A., Sethi, R., and Ullman, J. (1986). Compilers: Principles, techniques, and tools. In *Addison-Wesley series in computer science / World student series edition*. (Cité en pages 2, 4, 28, 32, 47, 51 et 55.)

[5] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Non cité.)

[6] Aiken, A. (2020). Compilers : Online course. In *http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=Compilers*. (Cité en pages 2, 34 et 51.)

[7] Ananian C. S., Flannery F., e. a. (2020). Cup : Construction of useful parsers. http://www2.cs.tum.edu/projects/cup/. (Cité en page 79.)

[8] Brzozowski, J. A. (1964). Derivatives of regular expressions. *J. ACM*, 11(4):481–494. (Cité en pages 38 et 40.)

[9] Chomsky, N. (1956). Three models for the description of language. *IRE Trans. Inf. Theory*, 2(3):113–124. (Cité en page 13.)

[10] Cooper, K. and Torczon, L. (2003). Engineering a compiler. (Cité en pages 2 et 28.)

[11] David Watt, Deryck Brown, R. W. S. (2007). *Programming Language Processors in Java : Compilers and Interpreters AND Concepts of Programming Languages*. Pearson international edition. Prentice Hall Press. (Cité en pages iii et 4.)

[12] DeRemer, F. (1971). Simple lr(k) grammars. *Commun. ACM*, 14(7):45–460. (Cité en page 72.)

[13] et al., W. R. (2013). Springer. (Cité en pages 47 et 69.)

[14] F., D. and J., P. T. (1979). Efficient computation of lalr(1) look-ahead sets. pages 176–187. In Johnson, S. C., editor, Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, Denver, Colorado, USA. (Cité en page 78.)

[15] Hanspeter Mössenböck, Markus Löberbauer, A. W. (2020). The compiler generator coco/r. http://www.ssw.uni-linz.ac.at/coco/. (Cité en page 79.)

[16] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2007). *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley. (Cité en pages 9, 24, 36, 51, 73 et 85.)

[17] Johnson, S. C. (1979). Yacc : Yet another compiler compiler. Technical report. (Cité en page 80.)

[18] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J., editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ. (Cité en pages 23 et 32.)

[19] Knuth, D. E. (1965). On the translation of languages from left to right. *Inf. Control.*, 8(6):607–639. (Cité en pages 68 et 69.)

[20] Lehman, E. (2017). *Mathematics for Computer Science*. Samurai Media Limited, London, GBR. (Cité en page 9.)

[21] Levine, J. R. (2009). *flex and bison - Unix text processing tools*. O'Reilly. (Cité en pages 32, 41, 79 et 80.)

[22] Lewis, P. M. and Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, 15(3):465–488. (Cité en page 59.)

[23] Lucas, P. (1961). The structure of formula-translators. *ALGOL Bull.*, (Suppl.16):1–27. (Cité en page 55.)

[24] M. E. Lesk, E. S. (1975). Lex - a lexical analyzer generator. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974. (Cité en page 41.)

[25] Owens, S., Reppy, J. H., and Turon, A. (2009). Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190. (Cité en page 38.)

[26] Rabin, M. O. and Scott, D. S. (1959). Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125. (Cité en page 18.)

[27] Reps, T. W. (1998). "maximal-munch" tokenization in linear time. *ACM Trans. Program. Lang. Syst.*, 20(2):259–273. (Cité en page 35.)

[28] Rosenkrantz, D. J. and Stearns, R. E. (1970). Properties of deterministic top-down grammars. *Inf. Control.*, 17(3):226–256. (Cité en page 59.)

[29] Scott, M. L. (2009). *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (Cité en pages iii, 2, 3, 4 et 6.)

[30] Stubblebine, T. (2003). *Regular expression - pocket reference: regular expressions for Perl, C, PHP, Python, Java, and .NET*. O'Reilly. (Cité en page 32.)

[31] Thompson, K. (1968). Regular expression search algorithm. *Commun. ACM*, 11(6):419–422. (Cité en page 38.)

[32] Tremblay, J. and Sorenson, P. (1985). The theory and practice of compiler writing. (Cité en page 2.)

[33] Viswanadha, S. and Sankar, S. (2020). Javacc, the most popular parser generator for use with java applications: https://javacc.github.io/javacc/faq.html. (Cité en page 79.)