



# ***COMPILATION***

3<sup>ème</sup> Année Licence en Informatique

***SUPPORT DE COURS REALISE PAR  
DR TOUFIK SARI  
PR LABIBA SOUICI-MESLATI***

toufik.sari@univ-annaba.dz  
labiba.souici@univ-annaba.org

***2020-2021***



**Domaine:** Mathématique et Informatique

**Spécialité:** Licence Informatique (L3)

**Unité d'enseignement:** UEI13

**Nombre de Crédits:** 6

**Volume horaire hebdomadaire total :** 04h30

**Filière:** Informatique

**Semestre:** 1, Année 2020-2021

**Matière:** COMPILATION

**Cours :** 1h30

**TD :** 1h30

**TP :** 1h30

**Evaluation**

**Examen final :** 50%

**TD :** 25% (50% : présence et participation et 50% : micro interrogation)

**TP :** 25% (50% : présence et efforts pendant les TP et 50% : évaluations sur machine)

## Objectifs

---

1. Compréhension du cheminement d'un programme (texte) source vers un programme (code).
2. Etude des étapes du processus de compilation d'un langage évolué.
3. Etude de méthodes et techniques utilisées en analyse lexicale, syntaxique et sémantique.
4. Familiarisation, en TP, avec des outils de génération d'analyseurs lexicaux et syntaxiques (LEX et YACC).

## Contenu

---

### 1 Introduction à la compilation

- Les différentes étapes de la compilation
- Compilation, interprétation, traduction

### 2 Analyse Lexicale

- Expressions régulières
- Grammaires
- Automates d'états finis
- Un exemple de générateur d'analyseurs lexicaux : LEX

### 3 Analyse Syntaxique

- Définitions : grammaire syntaxique, récursivité gauche, factorisation d'une grammaire, grammaire  $\epsilon$ -libre
- Calcul des ensembles des débuts et suivants
- Méthodes d'analyse descendante : la descente récursive, LL(1)
- Méthodes d'analyse ascendante : SLR(1), LR(1), LALR(1) (méthode des items)
- Un exemple de générateur d'analyseurs syntaxiques : YACC

### 4 Traduction dirigée par la syntaxe (Analyse sémantique)

### 5 Formes intermédiaires

- Forme postfixée
- Quadruplés
- Triplés directs et indirects
- Arbre abstrait

### 6 Allocation - Substitution – Organisation des données à l'exécution

## Références Bibliographiques

---

Ouvrages existants au niveau de la bibliothèque de l'université, la référence 1 est vivement recommandée

1. Aho A., Sethi R., Ullman J., "Compilateurs : Principes, techniques et outils", Inter-éditions, 1991 et Dunod, 2000
2. Drias H., "Compilation: Cours et exercices", OPU, 1993
3. Wilhem R., Maurer D., "Les compilateurs: Théorie, construction, génération", Masson, 1994

# CHAPITRE 1 : INTRODUCTION AUX COMPILATEURS

---

## 1.1 Définition

---

Un compilateur est un programme qui a comme entrée un code source écrit en langage de haut niveau (langage évolué) est produit comme sortie un code cible en langage de bas niveau (langage d'assemblage ou langage machine).

La traduction ne peut être effectuée que si le code source est correct car, s'il y a des erreurs, le rôle du compilateur se limitera à produire en sortie des messages d'erreurs (voir figure 1.1).

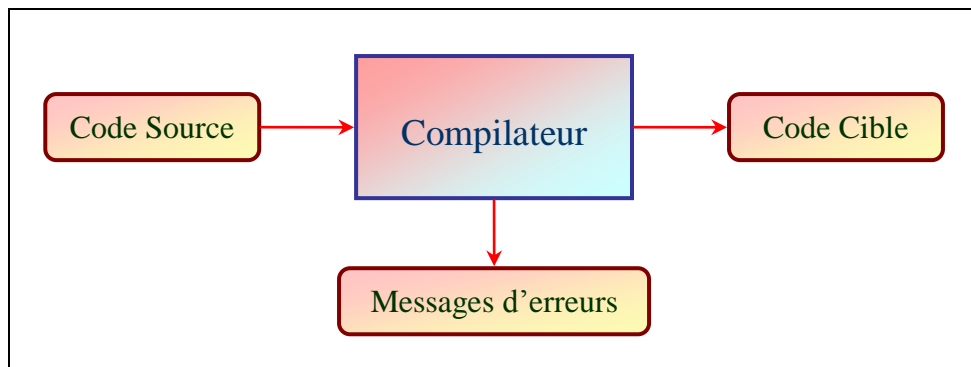


Figure 1.1. Rôle du compilateur

Un compilateur est donc un traducteur de langage évolué qu'on ne doit pas confondre avec un interpréteur qui est un autre type de traducteur. En effet, au lieu de produire un programme cible comme dans le cas d'un compilateur, un interpréteur exécute lui-même au fur et à mesure les opérations spécifiées par le programme source. Il analyse une instruction après l'autre puis l'exécute immédiatement. A l'inverse d'un compilateur, il travaille simultanément sur le programme et sur les données. L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté, ce qui n'est pas le cas avec un compilateur. Généralement, les interpréteurs sont assez petits, mais le programme est plus lent qu'avec un langage compilé. Un autre inconvénient des interpréteurs est qu'on ne peut pas cacher le code, et donc garder des secrets de fabrication : toute personne ayant accès au programme peut le consulter et le modifier comme elle le veut. Par contre, les langages interprétés sont souvent plus simples à utiliser et tolèrent plus d'erreurs de codage que les langages compilés. Des exemples de langages interprétés sont : BASIC, scheme, CaML, Tcl, LISP, Perl, Prolog

Il existe des langages qui sont à mi-chemin de l'interprétation et de la compilation. On les appelle langages P-code ou langages intermédiaires. Le code source est traduit (compilé) dans une forme binaire compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Lorsqu'on exécute le programme, ce P-code est interprété. Par exemple en Java, le programme source est compilé pour obtenir un fichier (.class) « byte code » qui sera interprété par une machine virtuelle. Un autre langage p-code : Python.

Les interpréteurs de p-code peuvent être relativement petits et rapides, si bien que le p-code peut s'exécuter presque aussi rapidement que du binaire compilé. En outre les langages p-code peuvent garder la flexibilité et la puissance des langages interprétés.

## 1.2 Structure générale d'un compilateur

Un compilateur est généralement composé de modules correspondant aux phases logiques de l'opération de compilation (voir figure 1.2).

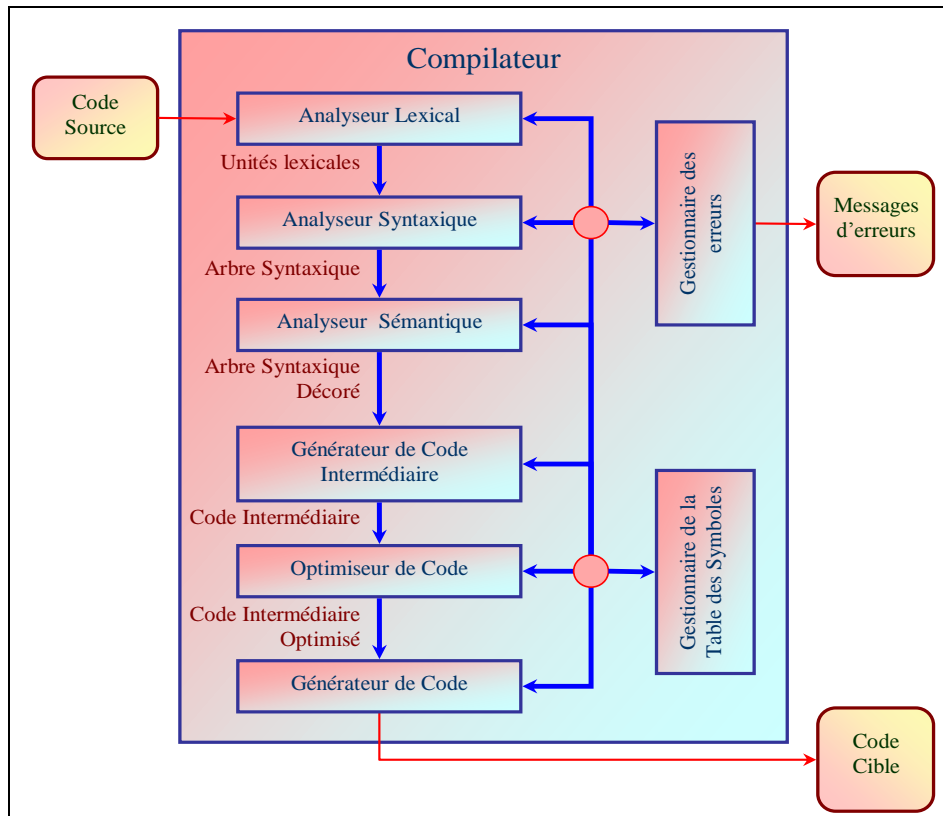


Figure 1.2. Phases et modules de compilation

Chacun des modules de la figure 1.2 (excepté les modules gestionnaires de table de symboles et d'erreurs), représente une phase logique qui reçoit en entrée une représentation de code source et la transforme en une autre forme de représentation.

La structure représentée par la figure 1.2 est purement conceptuelle. Elle correspond à l'organisation logique d'un compilateur. En pratique, plusieurs phases peuvent être regroupées en une seule passe qui reçoit en entrée une certaine représentation et donne en sortie une autre.

Par exemple, les phases d'analyses lexicale, syntaxique, sémantique et la génération du code intermédiaire peuvent correspondre à seule passe dans laquelle l'analyseur syntaxique est le module maître qui appelle, à chaque fois, l'analyseur lexical pour obtenir une unité lexicale, puis déterminer graduellement la structure syntaxique du code et enfin appelle le générateur de code qui effectue l'analyse sémantique et produit une partie du code intermédiaire.

### 1.2.1 L'analyseur lexical

Connu aussi sous l'appellation **Scanner**, l'analyseur lexical a pour rôle principal la lecture du texte du code source (suite de caractères) puis la formation des unités lexicales (appelées aussi entités lexicales, lexèmes, jetons, tokens ou encore atomes lexicaux).

#### Exemple

Considérons l'expression d'affectation **a := b + 2 \* c ;**

Les unités lexicales qui apparaissent dans cette expression sont :

Unité lexicale	Sa nature
a	Identificateur de variable
:=	Symbole d'affectation
b	Identificateur de variable
+	Opérateur d'addition
2	Valeur entière
*	Opérateur de multiplication
c	Identificateur de variable
;	Séparateur

L'analyseur a aussi comme rôle l'élimination des informations inutiles pour l'obtention du code cible, le stockage des identificateurs dans la table des symboles et la transmission d'une entrée à l'analyseur syntaxique. Concernant les informations inutiles, il s'agit généralement du caractère espace et des commentaires.

### 1.2.2 L'analyseur syntaxique

L'analyseur syntaxique (appelé Parser en anglais) a pour rôle principal la vérification de la syntaxe du code en regroupant les unités lexicales suivant des structures grammaticales qui permettent de construire une représentation syntaxique du code source. Cette dernière a souvent une structure en arbre. Notons que durant cette phase, des informations, telles que le type des identificateurs, sont enregistrées dans la table des symboles

#### Exemple

La représentation sous forme d'arbre syntaxique de l'expression « **a := b + 2 \* c ;** » est donnée par la figure 1.3. Dans cette structure d'arbre, les nœuds représentent des opérateurs et les feuilles de l'arbre représentent les valeurs et les variables sur lesquelles s'effectuent les opérations. La figure donne aussi le parcours qui est fait sur cet arbre lors de l'évaluation. D'autres parcours peuvent être envisagés pour réaliser différentes tâches, cependant ces parcours ont lieu dans les phases ultérieures de la compilation.

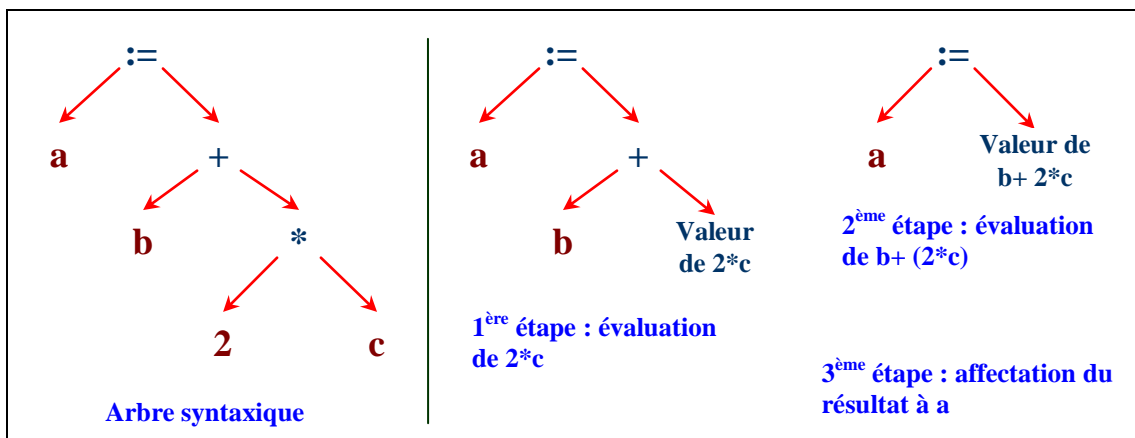


Figure 1.3. Arbre syntaxique et parcours d'évaluation

### 1.2.3 L'analyseur sémantique

Il comme rôle principal le contrôle du code source, pour détecter éventuellement l'existence d'erreurs sémantiques, et la collecte des informations destinées à la production du code intermédiaire. Un des constituants importants de la phase d'analyse sémantique est le contrôle du type qui consiste à vérifier si les opérandes de chaque opérateur sont conformes aux spécifications du langage utilisé pour le code source.

#### Exemple

Dans l'analyse sémantique de « **a := b + 2 \* c ;** », il faut vérifier que, si a est de type entier, alors b et c le sont aussi, sinon il faut signaler une erreur.

Si on suppose que *a*, *b* et *c* sont de type réel, alors pendant l'évaluation de l'expression, l'analyse sémantique aura comme tâche d'insérer une opération de conversion de type pour transformer la valeur entière 2 en valeur réelle 2.0. Cela peut être effectué sur l'arbre syntaxique comme le montre la figure 1.4 (arbre syntaxique décoré).

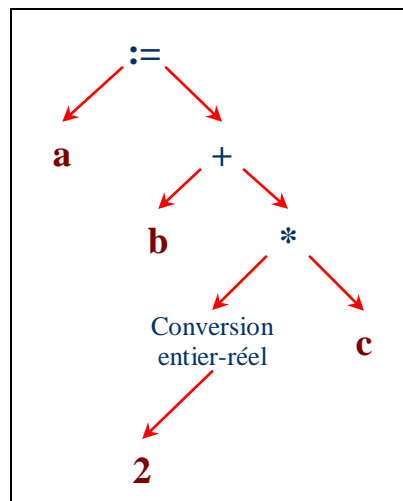


Figure 1.4. Enrichissement de l'arbre syntaxique lors de la phase d'analyse sémantique

### 1.2.4 Le générateur de code intermédiaire

Certains compilateurs construisent explicitement une représentation intermédiaire du code source sous forme d'un code intermédiaire qui n'est pas directement exécutable par une machine spécifique. C'est plutôt un code généré pour une machine abstraite (virtuelle), qui a la double caractéristique d'être, à la fois, facile à produire, à partir de l'arbre syntaxique, et facile à convertir pour une machine réelle donnée.

#### Exemple

Nous avons supposé que la machine abstraite est une machine à une adresse qui dispose d'un seul accumulateur et dont le jeu d'instruction contient les instructions LoadValue, ConvReal, Mul, Add et Store. Elles permettent de réaliser les opérations données dans la deuxième colonne. Nous avons supposé aussi que *a* occupe la première entrée dans la table des symboles, *b* occupe la deuxième et *c* la troisième.

Ainsi, le code intermédiaire de l'expression « *a := b + 2 \* c ;* », peut être comme suit :

Code	Signification opérationnelle des instructions
<b>LoadValue 2</b>	Charger l'accumulateur avec une valeur directe (2)
<b>ConvReal</b>	Convertir le contenu de l'accumulateur en réel
<b>Store temp1</b>	Stocker le résultat dans la variable temporaire temp1
<b>Load temp1</b>	Charger l'accumulateur avec la valeur de temp1
<b>Mul 3</b>	Multiplier le contenu de l'accumulateur par le contenu de l'entrée 3 de la table de symboles (c)
<b>Add 2</b>	Additionner le contenu de l'accumulateur au contenu l'entrée 2 de la table de symboles (b)
<b>Store 1</b>	Ranger le contenu de l'accumulateur dans l'entrée 1 de la table de symboles (a)

### 1.2.5 L'optimiseur du code intermédiaire

Lors de la phase d'optimisation, le code intermédiaire est changé pour améliorer les performances du code cible qui en sera généré. Il s'agit principalement de réduire le temps d'exécution et l'espace mémoire qui sera occupé par le code cible. L'optimisation supprime, par exemple, les identificateurs non utilisés, élimine les instructions inaccessibles, élimine les instructions non nécessaires, fait ressortir hors des boucles les instructions qui ne dépendent pas de l'indice de parcours des boucles, etc.

L'optimisation risque de ralentir le processus de compilation dans son ensemble mais elle peut avoir un effet positif considérable sur le code cible qui sera généré ultérieurement.

### Exemple

On constate dans l'exemple précédent que la valeur convertie 2.0 est stockée dans temp1 puis récupérée et chargée dans l'accumulateur. Puisque aucun usage n'est fait de temp1 dans le reste du code, il est possible d'éliminer les deux instructions en question. Après conversion, le résultat 2.0 reste alors dans l'accumulateur et sera utilisé directement dans la multiplication. Noter que, à l'opposé, si la conversion en réel de la valeur 2 est souvent nécessaire, il serait préférable de la stocker dans un espace temporaire. Cependant, ce dernier augmente l'espace réservé aux données dans le code cible.

Le nouveau code intermédiaire est le suivant :

Code	Signification opérationnelle des instructions
<b>LoadValue 2</b>	Charger l'accumulateur avec une valeur directe
<b>ConvReal</b>	Convertir le contenu de l'accumulateur en réel
<b>Mul 3</b>	Multiplier le contenu de l'accumulateur par le contenu de l'entrée 3 de la table de symboles
<b>Add 2</b>	Additionner le contenu de l'accumulateur au contenu l'entrée 2 de la table de symboles
<b>Store 1</b>	Ranger le contenu de l'accumulateur dans l'entrée 1 de la table de symboles

### 1.2.6 Le générateur du code cible

C'est la phase finale d'un compilateur qui consiste à produire du code cible dans un langage d'assemblage ou un langage machine donné. Le code généré est directement exécuté par la machine en question ou alors il l'est après une phase d'assemblage.

#### Exemple

Considérons une machine à deux adresses qui dispose de deux registres de calcul R1 et R2. Nous supposons que les variables a, b et c de la table des symboles ont comme adresses de cellules mémoires correspondantes ad1, ad2 et ad3. Le code cible qui sera généré pour cette machine est donné dans le tableau suivant :

Code	Signification opérationnelle des instructions
<b>MOV R1, #2</b>	Charger R1 avec la valeur directe 2
<b>CReal R1</b>	Convertir le contenu de R1 en réel
<b>MOV R2, ad3</b>	Charger le registre R2 avec le contenu de la cellule mémoire d'adresse ad3
<b>MUL R1, R2</b>	Multiplier le contenu de R1 par le contenu de R2, le résultat est dans le premier registre
<b>MOV R2, ad2</b>	Charger R2 avec le contenu de l'adresse ad2
<b>ADD R1, R2</b>	Additionner le contenu de R1 au contenu de R2, le résultat est dans le premier registre
<b>STO R1, ad1</b>	Ranger le contenu de R1 dans la cellule mémoire d'adresse ad1

### 1.2.7 Le gestionnaire de la table de symbole

Les phases logiques de compilation échangent des informations par l'intermédiaire de la table des symboles. C'est une structure de données (généralement une table) contenant un enregistrement pour chaque identificateur utilisé dans le code source en cours d'analyse. L'enregistrement contient, parmi d'autres informations, le nom de l'identificateur, son type, et l'emplacement mémoire qui lui correspondra lors de l'exécution.

A chaque fois que l'analyseur lexical rencontre un identificateur pour la première fois, le gestionnaire de la table des symboles insère un enregistrement dans la table et l'initialise avec les informations actuellement disponibles (le nom). Lors de l'analyse syntaxique, le gestionnaire associera le type à l'identificateur, alors que, lors de l'analyse sémantique, une vérification de types est opérée grâce à cet enregistrement.

### 1.2.8 Le gestionnaire des erreurs

Son rôle est de signaler les erreurs qui peuvent exister dans le code source et qui sont détectées lors des différentes phases logiques de la compilation. Il doit produire, pour chaque erreur, un diagnostic clair et sans ambiguïté qui permettra la localisation et la correction de l'erreur par l'auteur du code source.



## ***1.3 Outils de construction des compilateurs***

---

Suite au développement des premiers compilateurs, on s'est très vite rendu compte que certaines tâches liées au processus de compilation peuvent être automatisées, ce qui facilite grandement la construction des compilateurs. La notion d'outils de construction de compilateurs est alors apparue.

La première catégorie est représentée par des outils généraux appelés Compilateurs de Compilateurs, Générateurs de Compilateurs ou Systèmes d'écriture de traducteurs. Les outils de cette catégorie se sont souvent orientés vers un modèle particulier de langages et ne sont adaptés qu'à la construction de compilateurs pour des langages correspondant à ce modèle.

La deuxième catégorie correspond à des outils spécialisés dans la construction automatique de certaines phases d'un compilateur, tels que :

- Les constructeurs ou générateurs automatiques d'analyseurs lexicaux à partir d'une spécification contenant des expressions régulières
- Constructeurs ou générateurs automatiques d'analyseurs syntaxiques à partir d'une spécification basée sur une grammaire non contextuelle et en utilisant des algorithmes d'analyse puissants mais difficile à mettre en œuvre manuellement.

### 2.1 Rôle de l'analyseur lexical

Il s'agit de transformer des suites de caractères du code source en suite de symboles, correspondant aux unités lexicales, que l'analyseur lexical produit comme résultat de l'analyse. Pour cela, il existe deux approches possibles sachant qu'une passe est définie comme correspondant à un traitement entier d'une représentation du programme source pour produire une représentation équivalente en mémoire secondaire :

- L'analyseur lexical effectue un traitement de la totalité du code source en une première passe, ce qui lui permet d'en obtenir une représentation équivalente sous forme d'une suite d'unités lexicales, sauvegardée dans un fichier en mémoire secondaire. Ce fichier sera l'entrée de l'analyseur syntaxique, qui accomplira son travail pendant une deuxième passe. Le schéma de la figure 2.1 illustre cette approche.
- L'analyseur lexical fonctionne comme une procédure sollicitée à chaque fois, par l'analyseur syntaxique, pour lui fournir une unité lexicale. L'analyseur lexical effectue, pour cela, son travail, pas à pas, en synchronisation avec l'avancement de l'analyse syntaxique. On dit que l'analyse lexicale et syntaxique partagent une même passe. Dans ce cas, il n'est plus question de sauvegarder les unités dans un fichier intermédiaire, cependant, les deux analyseurs (lexical et syntaxique) doivent se trouver ensemble en mémoire. Cette approche est la plus utilisée dans la conception des compilateurs, elle est illustrée par la figure 2.2.

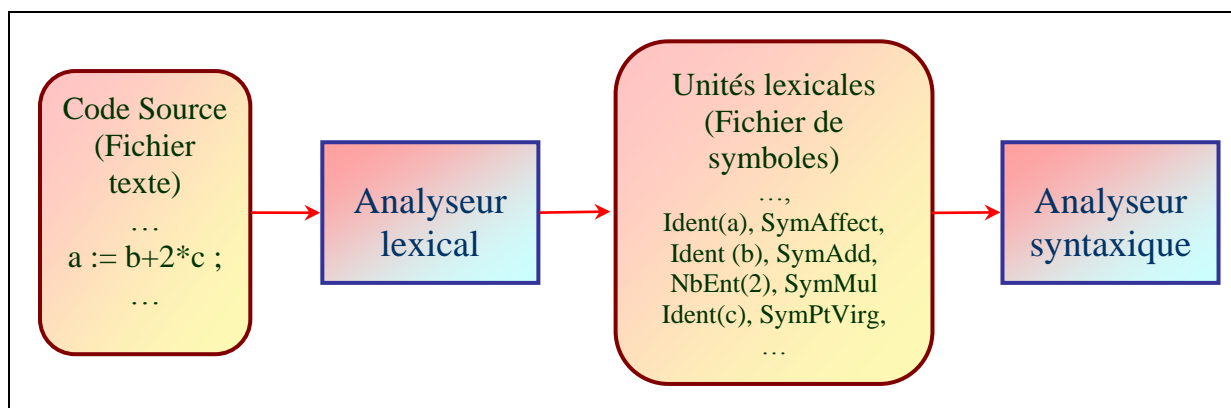


Figure 2.1. Analyse lexicale et syntaxique en deux passes différentes

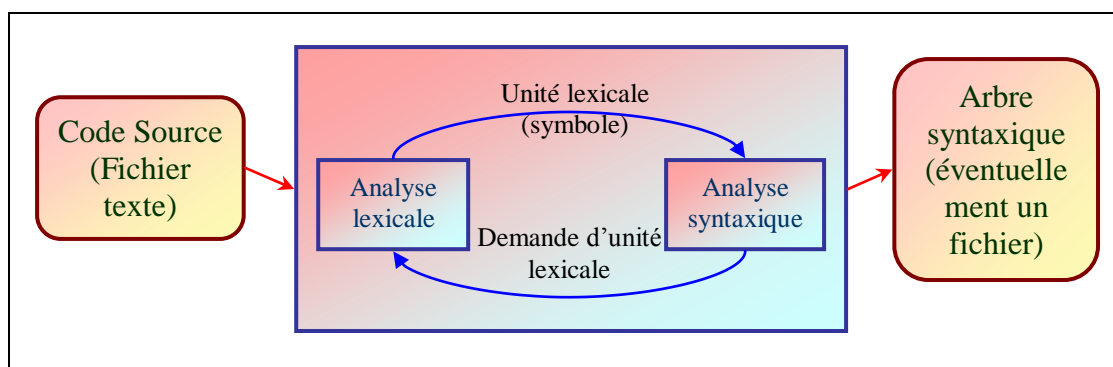


Figure 2.2. Analyses lexicale et syntaxique en une seule passe

### 2.2 Tâches effectuées par l'analyseur lexical

L'analyse lexicale effectue un ensemble de tâches :

- 1- Lecture du fichier du code source (fichier texte) caractère par caractère avec éventuellement l'élimination de caractères et informations inutiles (blancs, tabulations, commentaires, caractère de fin de ligne, ...) pour réduire la représentation du programme. Une erreur est signalée à ce niveau si un caractère non permis (illégal) par le langage est rencontré.

- 2- Concaténation des caractères pour former des unités lexicales et signaler, éventuellement, une erreur si la chaîne dépasse une certaine taille.
- 3- Association d'un symbole à chaque unité lexicale. Cette opération peut engendrer une erreur si l'unité formée ne correspond à aucune unité légale du langage.
- 4- Enregistrement de chaque unité lexicale et des informations la concernant (nom, valeur, ...) dans la table des symboles (en invoquant le gestionnaire de la table des symboles) ou dans un fichier résultat. A ce niveau, il est possible de détecter certaines erreurs telles que la double déclaration d'un identificateur, l'utilisation d'un identificateur sans déclaration préalable, etc.
- 5- Création d'une liaison entre les unités lexicales d'une ligne et la ligne du fichier la contenant. Cette opération, qui se fait par une simple numérotation des lignes du texte du code source, permet la localisation des lignes en cas d'erreur. Dans certains compilateurs, l'analyseur lexical est chargé de créer une copie du programme source en y intégrant les messages d'erreurs lexicales.

Les symboles associés aux unités lexicales correspondent à la nature de celles-ci. On distingue plusieurs catégories de symboles :

- 1- Les mots réservés du langage (if, then, begin pour le langage Pascal, par exemple)
- 2- Les constantes (3, 3.14, True, 'Bonjour', ...)
- 3- Les identificateurs qui peuvent être des noms de variables, de fonctions, de procédures, etc.
- 4- Les symboles spéciaux, en tenant compte des possibilités d'assemblage de caractères tels que < > <= > : := + - /\*
- 5- Les séparateurs tels que ; (point virgule) et . (point).

## 2.3 Spécification des unités lexicales

L'ensemble des unités lexicales, qu'un analyseur reconnaît, constitue un langage régulier L qui peut être décrit par des expressions régulières et reconnu par un automate d'états fini.

La spécification ou description des unités lexicales peut donc être effectuée en utilisant une notation appelée **expressions régulières**. Une expression régulière est construite à partir d'expressions régulières plus simples, en utilisant un ensemble de règles de définition qui spécifient comment le langage est formé.

Soit le vocabulaire  $\Sigma = \{a, b, c\}$

- 1-  $\varepsilon$  dénote la chaîne vide
- 2- L'expression régulière  $a/b$  dénote l'ensemble des chaînes  $\{a, b\}$
- 3- L'expression  $a^*$  dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque de  $a$  (pouvant être nul),  $c$  à  $d \in \{\varepsilon, a, aa, aaa, \dots\}$
- 4- L'expression  $a^+$  dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque, non nul, de  $a$ ,  $c$  à  $d \in \{a, aa, aaa, \dots\}$ . Si  $r$  est une expression régulière alors  $r^* = r^+/\varepsilon$  et  $r^+ = rr^*$
- 5- La notation  $r?$  est une abréviation de  $r/\varepsilon$  et signifie zéro ou une instance de  $r$
- 6- La notation  $[abc]$ , où  $a, b, c$  sont des symboles de l'alphabet, dénote l'expression régulière  $a/b/c$ . Une classe de caractères  $[a-z]$  signifie  $a/b/c/d/ \dots/z$

### Exemple 1

Pour chacune des expressions régulières  $r_i$  suivantes, on souhaite déterminer le langage dénoté par  $r_i$ .

$$\begin{aligned} r_1 &= (a/b)(a/b) \\ r_2 &= (a/b)^* \\ r_3 &= (a^*b^*)^* \\ r_4 &= a/a^*b \end{aligned}$$

Chaque langage dénoté par  $r_i$  est noté  $L(r_i)$

$$L(r_1) = \{aa, ab, ba, bb\}$$

$L(r_2) = \{\varepsilon, a, aa, aaa, \dots, b, bb, bbb, \dots, ab, aab, \dots, abb, abbb, ba, bba, bbaa, aba, \dots\}$ . C'est l'ensemble de toutes les chaînes composées d'un nombre quelconque de  $a$  et d'un nombre quelconque de  $b$ .

$$L(r3) = L(r2)$$

$L(r4) = \{a, b, ab, aab, aaab, \dots, aaaa\dots ab\}$  En plus de la chaîne a, cet ensemble contient des chaînes composées d'un nombre quelconque non nul de a suivis par un b.

### Exemple 2

On souhaite déterminer la définition régulière de l'ensemble des identificateurs d'un langage sachant qu'ils sont constitués de suite de lettres et de chiffres commençant par une lettre.

Cette définition régulière peut être donnée par :

- lettre = A/B/.../Z/a/b/.../z ou bien [A-Za-z]
- chiffre = 0/1/2/3/4/5/6/7/8/9 ou bien [0-9]
- ident = lettre (lettre/chiffre)\* qu'on peut noter directement ident = [A-Za-z] [A-Za-z0-9]\*

### Remarque

Les expressions régulières sont un outil puissant et pratique pour définir les constituants élémentaires des langages de programmation. Cependant, le pouvoir descriptif des expressions régulières est limité car elles ne peuvent pas être utilisées dans certains cas tels que :

- ❖ La définition des constructions équilibrées ou imbriquées. Par exemple, pour les chaînes contenant obligatoirement pour chaque parenthèse ouvrante une parenthèse fermante, les expressions régulières ne permettent pas d'assurer cela, on dit qu'elles **ne savent pas compter**.
- ❖ La définition d'expressions contenant des répétitions de sous chaînes à différentes positions (par exemple des chaînes de la forme  $\alpha\alpha\alpha\alpha$  où  $\alpha$  est une combinaison des caractères a et b).
- ❖ La définition d'expressions contenant à différentes positions des séquences de caractères ayant la même longueur (par exemple  $a^n b^n$  où n désigne le nombre de répétitions du caractère qui le précède).

On peut, cependant, utiliser les expressions régulières dans le cas d'un nombre **fixe** de répétitions d'une construction donnée.

## 2.4 Approches de construction des analyseurs lexicaux

---

Après la description des unités lexicales, la construction de l'analyseur lexical peut être effectuée selon l'une des manières suivantes :

- 1- Une approche simplifiée (manuelle) basée sur les diagrammes de transition.
- 2- Une approche plus rigoureuse basée sur les automates d'états finis.
- 3- Une approche utilisant un outil générateur d'analyseurs lexicaux tel que Lex, FLex, JFLex...

Dans les sections suivantes nous décrivons principalement l'approche basée sur les automates à états finis.

## 2.5 Construction d'analyseur lexical basée sur les automates finis

---

### 2.5.1 Démarche générale de construction d'un analyseur lexical

---

En se basant sur les propositions démontrables suivantes :

- L'ensemble des unités lexicales d'un langage donné constitue un langage régulier L.
- Pour toute expression régulière r, il existe un automate fini non déterministe (AFN) qui accepte l'ensemble régulier décrit par r.
- Si un langage L est accepté par un automate fini non déterministe (AFN), alors il existe automate fini déterministe (AFD) acceptant L.

On peut définir une approche rigoureuse pour la construction d'un analyseur lexical en utilisant les automates d'états finis. Cette approche est constituée de 6 étapes :

**Etape 1 : Spécification des unités lexicales**

Spécifier chaque type d'unité lexicale à l'aide d'une expression régulière (ER).

**Etape 2 : Conversion ER en AFN**

Convertir chaque expression régulière en un automate d'états fini (non déterministe).

**Etape 3 : Réunion des AFNs**

Construire l'automate Union de tous les automates de l'étape 2

(on peut ajouter un nouvel état initial d'où partent un ensemble d'arcs étiquetés  $\epsilon$ ).

**Etape 4 : Détermination ou transformation de l'AFN obtenu en AFD**

Rendre l'automate de l'étape 3 déterministe.

**Etape 5 : Minimisation de l'AFD.**

Minimiser l'automate obtenu à l'étape 4.

**Etape 6 : Implémentation de l'AFD minimisé.**

Implémenter l'automate obtenu à l'étape 5 en le simulant à partir de sa table de transition.

## 2.5.2 Automates à états finis

On transforme une expression régulière en un reconnaiseur en construisant un diagramme de transition généralisé appelé automate à états finis. Ce dernier peut être déterministe (AFD) ou non déterministe (AFN).

	<b>AFN</b>	<b>AFD</b>
<b>Temps de reconnaissance</b>	Augmente (à cause des retours arrière)	Diminue (un seul chemin possible pour une chaîne donnée)
<b>Espace occupé</b>	Nombre d'états plus petit que celui de l'AFD équivalent	Nombre d'états généralement plus grand que celui de l'AFN équivalent

La transformation d'un automate à états finis en un analyseur lexical consiste à associer une unité lexicale à chaque état final et à faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question. Pour cela il faut implémenter la fonction de transition de l'automate en utilisant une table de transition.

### 2.5.2.1 Automate fini non déterministe

Un AFN est défini par :

- Un ensemble d'état  $E$
- Un ensemble de symboles d'entrée ou alphabet  $\Sigma$
- Un état initial  $e_0$
- Un ensemble  $F$  d'états finaux ou d'acceptation
- Une fonction de transition *Transiter* qui fait correspondre à chaque couple (état,symbole), un ensemble d'états.

**Exemple**

Considérons l'AFN qui reconnaît le langage décrit par l'expression régulière  $(a|b)^*abb$  (voir figure 2.3). Pour cet automate :

$$E = \{0, 1, 2, 3\} \quad e_0 = 0 \quad \Sigma = \{a, b\} \quad F = \{3\}$$

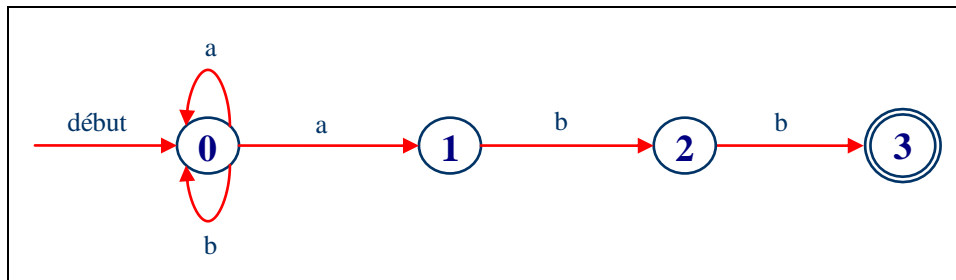


Figure 2.3. Un AFN pour l'expression  $(a|b)^*abb$

L'implémentation la plus simple d'un AFN est une table de transition dans laquelle il y a une ligne pour chaque état et une colonne pour chaque symbole d'entrée (avec la chaîne vide  $\epsilon$  si nécessaire). L'entrée pour la ligne  $i$  et le symbole  $a$  donne l'ensemble des états (ou un pointeur vers cet ensemble) qui peuvent être atteints à partir de l'état  $i$  avec le symbole  $a$ . Pour l'AFN de la figure 2.3, la table de transition est la suivante (sachant que la ligne 3 peut être éliminée) :

Etats	Symboles	
	a	b
0	{0, 1}	{0}
1	/	{2}
2	/	{3}
3	/	/

Un AFN accepte une chaîne si et seulement s'il existe au moins un chemin correspondant à cette chaîne entre l'état initial et l'un des états finaux. Un chemin peut être représenté par une suite de transitions appelées *déplacements*. Par exemple, pour la reconnaissance de la chaîne *aabb* par l'AFN de la figure 2.3, il existe deux chemins possibles. Le premier conduit à l'acceptation de la chaîne alors que le second stationne sur l'état 0 qui n'est pas un état final (voir figures 2.4 et 2.5).

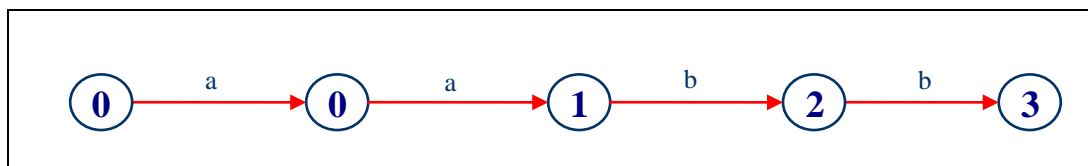


Figure 2.4. Déplacements réalisés en acceptant la chaîne *aabb*

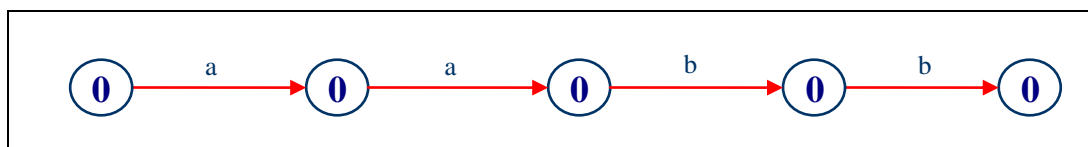


Figure 2.5. Stationnement sur l'état 0 pour la chaîne *aabb*

### 2.5.2.2 Automate fini déterministe

Un AFD est un cas particulier d'AFN dans lequel :

- Aucun état n'a de  $\epsilon$ -transition
- Pour chaque état  $e$  et chaque symbole d'entrée  $a$  il y a au plus un arc étiqueté  $a$  qui quitte  $e$

Dans la table de transition d'un AFD, une entrée contient un état unique au maximum (les symboles d'entrée sont les caractères du texte source), il est donc très facile de déterminer si une chaîne est acceptée par l'automate vu qu'il n'existe, au plus, qu'un seul chemin entre l'état initial et un état final étiqueté par la chaîne en question.

**Remarque :** La table de transition d'un AFN pour un modèle d'expression régulière peut être considérablement plus petite que celle d'un AFD. Cependant, l'AFD présente l'avantage de pouvoir reconnaître des modèles d'expressions régulières plus rapidement que l'AFN équivalent.

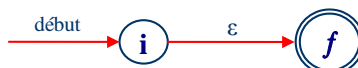
### 2.5.3 Construction d'AFN à partir d'expressions régulières (étape 2)

Il existe plusieurs algorithmes pour effectuer cette construction. Parmi les plus simples et les plus faciles à implémenter on cite l'algorithme de construction de **Thompson**. Dans la suite, nous utiliserons les notations suivantes :

- $r$  : désigne une expression régulière
- $N(r)$  : l'AFN correspondant à  $r$  (qui reconnaît  $r$ )

Les règles de l'algorithme de Thompson sont les suivantes :

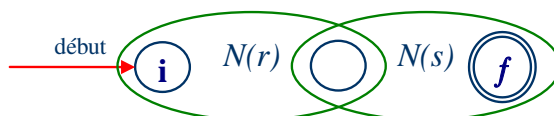
1. Pour  $r = \epsilon$   $N(\epsilon)$  est représenté par:



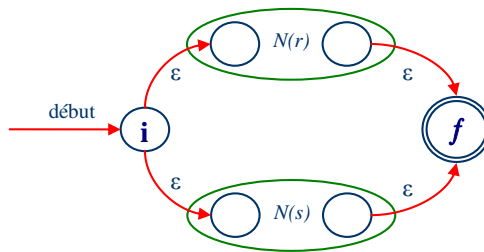
2. Pour  $r = a$   $N(a)$  est représenté par:



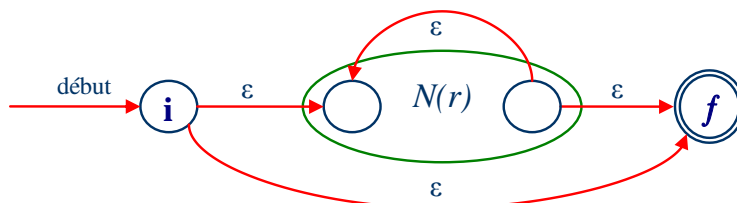
3. Pour  $N(rs)$  on a :



4. Pour  $N(r/s)$  on a :



5. Pour  $r^*$ ,  $N(r^*)$  sera donnée par la représentation suivante :



#### Exemple

L'AFN qui reconnaît l'expression régulière  $(a|b)^*abb$  selon la méthode de construction de Thompson est donné dans la figure 2.6 :

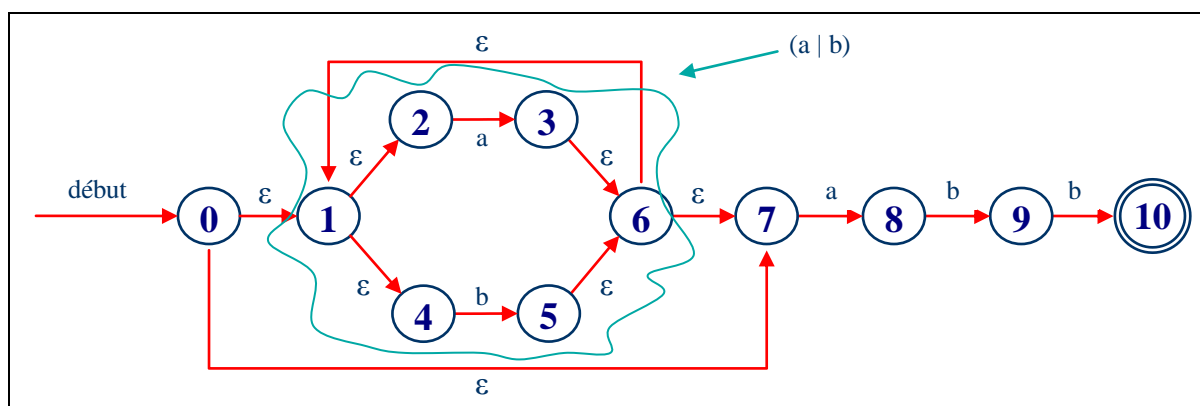


Figure 2.6. Automate fini non déterministe N pour accepter  $(a|b)^*abb$

## 2.5.4 Détermination d'un AFN (étape 4)

La détermination consiste à transformer un AFN en un AFD équivalent. Elle peut être effectuée par différentes méthodes, nous en présentons trois dans ce qui suit.

### 2.5.4.1 Détermination d'un AFN ne contenant pas de $\varepsilon$ -transitions

La détermination d'un AFN ne contenant pas de  $\varepsilon$ -transitions peut être effectuée en appliquant l'algorithme suivant sur la table de transition de l'AFN pour en déduire celle de l'AFD équivalent :

Etapes de l'algorithme

- 1- Partir de l'état initial  $E_0 = \{e_0\}$
- 2- Construire  $E_1$  qui est l'ensemble des états obtenus à partir de  $E_0$  par la transition étiquetée **a**,  
 $E_1 = \text{Transiter}(E_0, a)$ .
- 3- Recommencer l'étape 2 pour toutes les transitions possibles et pour chaque nouvel ensemble d'états  $E_i$ .
- 4- Tous les ensembles d'états  $E_i$  contenant au moins un état final deviennent finaux.
- 5- Renuméroter les ensembles d'états obtenus en tant que simples états.

#### Exemple

Appliquons l'algorithme précédent sur l'AFN dont  $e_0 = 0$  et  $F = \{2, 3\}$  et ayant la table de transition suivante :

Etats	Symboles	
	a	b
0	{0, 2}	{1}
1	{3}	{0, 2}
2	{3, 4}	{2}
3	{2}	{1}
4	/	{3}

Après application de l'algorithme, nous obtenons la table de transition de l'AFD suivante :

Etats	Symboles	
	a	b
{0}	{0, 2}	{1}
{0, 2}	{0, 2, 3, 4}	{1, 2}
{1}	{3}	{0, 2}
{0, 2, 3, 4}	{0, 2, 3, 4}	{1, 2, 3}
{1, 2}	{3, 4}	{0, 2}
{3}	{2}	{1}
{1, 2, 3}	{2, 3, 4}	{0, 1, 2}
{3, 4}	{2}	{1, 3}
{2}	{3, 4}	{2}
{2, 3, 4}	{2, 3, 4}	{1, 2, 3}
{0, 1, 2}	{0, 2, 3, 4}	{0, 1, 2}
{1, 3}	{2, 3}	{0, 1, 2}
{2, 3}	{2, 3, 4}	{1, 2}

Après renumérotation la table de l'AFD devient :

Etats	Symboles	
	a	b
A	B	C
B	D	E
C	F	B
D	D	G
E	H	B
F	I	C
G	J	K
H	I	L



I	H	I
J	J	G
K	D	K
L	M	K
M	J	E

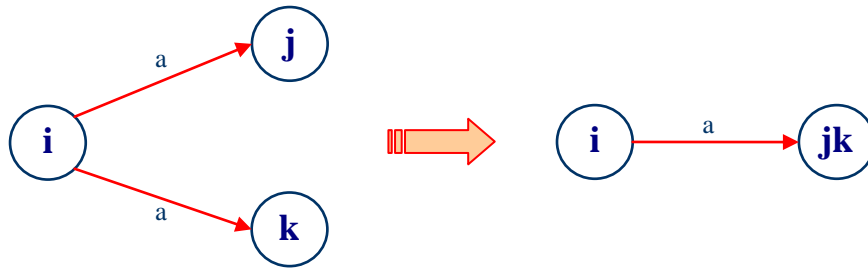
### 2.5.4.2 Transformation manuelle d'un AFN en AFD (cas général)

On utilise les règles de conversion suivantes pour rendre déterministe un AFN pouvant contenir des  $\epsilon$ -transitions:

1. Si on peut passer d'un état  $i$  à l'état  $j$  avec une transition étiquetée  $\epsilon$  alors les états  $i$  et  $j$  appartiennent à la même classe.



2. Si l'état  $i$  mène à l'état  $j$  et l'état  $k$  par des transitions identiques (même étiquette), alors les états  $j$  et  $k$  appartiennent à la même classe.



3. Une classe d'état est finale dans l'AFD si elle contient au moins un état final de l'AFN.

Ainsi, l'AFD équivalent à un AFN original possède des états qui sont des classes (regroupements) d'états de l'automate original. Les nœuds de l'AFD correspondent à des sous ensembles de nœuds de l'AFN qui ne sont pas forcément disjoints.

Si l'AFN à  $n$  nœuds, l'AFD peut en avoir  $2^n$ . Donc l'AFD peut être beaucoup plus volumineux que l'AFN équivalent, cependant, en pratique le nombre d'états est souvent plus petit que  $2^n$ .

#### Exemple

Considérons l'automate non déterministe de la figure 2.7, qui reconnaît l'ensemble des chaînes  $\{a, aa, ab, aba, abb\}$ .

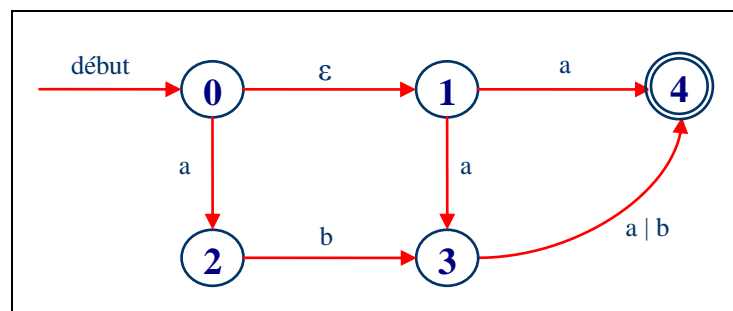


Figure 2.7. Exemple d'AFN

L'AFD correspondant est donné par la figure 2.8, on constate que les états ne sont pas disjoints (234, 34, 4).

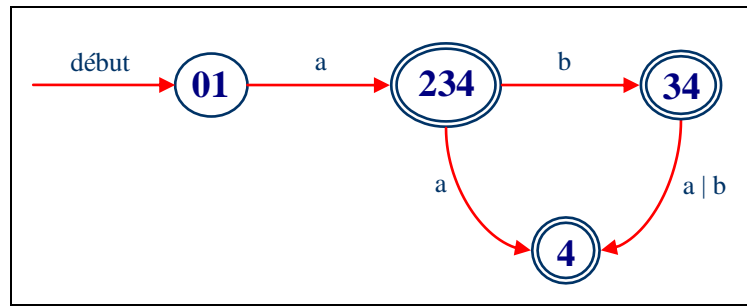


Figure 2.8. AFD équivalent à l'AFN de la figure 2.7

### 2.5.4.3 Méthode de construction de sous-ensembles pour la transformation des AFN en AFD (cas général)

La construction manuelle décrite précédemment (voir section 2.6.4.2) est simple, mais difficile à appliquer lorsque le nombre d'états est important. Un algorithme plus formel peut être utilisé pour cette étape. Ce dernier est connu sous le nom d'algorithme de *construction de sous-ensembles* et permet de construire une table de transition de l'AFD en se basant sur celle de l'AFN équivalent.

#### a- Notations et opérations utilisées

Soit un AFN noté  $N$  et  $D$  son AFD équivalent.

- $DTrans$  est la table des transitions de  $D$
- $Détats$  est l'ensemble des états de  $D$
- $e$  est un état de  $N$ ,  $e_0$  est l'état initial de  $N$
- $T$  est un ensemble d'états de  $N$
- L'opération  $\varepsilon$ -fermeture( $e$ ) est l'ensemble des états de  $N$  accessibles à partir de l'état  $e$  par des  $\varepsilon$ -transitions uniquement.

#### Remarque

Un état est accessible à partir de lui-même par une  $\varepsilon$ -transition (même si l'arc n'est pas visible).

- L'opération  $\varepsilon$ -fermeture( $T$ ) donne l'ensemble des états de  $N$  accessibles à partir des états  $e$  ( $e \in T$ ) par des  $\varepsilon$ -transitions uniquement.

Dans l'AFN représenté par la figure 2.9,  $\varepsilon$ -fermeture( $1$ ) =  $\{1, 2, 3, 4, 8\}$ ,  $\varepsilon$ -fermeture( $2$ ) =  $\{2, 3, 4\}$ ,  $\varepsilon$ -fermeture( $\{2, 5\}$ ) =  $\{2, 3, 4, 5, 6, 7\}$ ,  $\varepsilon$ -fermeture( $\{3, 6\}$ ) =  $\{3, 4, 6, 7\}$

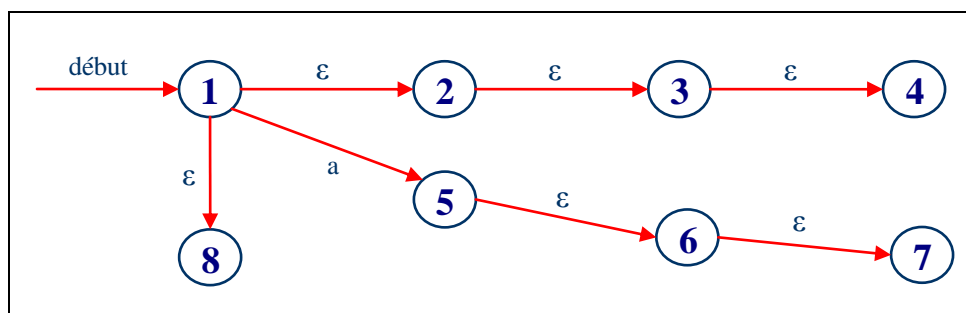


Figure 2.9. Exemple d'AFN pour le calcul  $\varepsilon$ -fermeture

- L'opération  $Transiter(T, a)$  donne l'ensemble des états de  $N$  vers lesquels il existe une transition avec le symbole d'entrée  $a$  à partir des états  $e \in T$ .

Dans l'AFN représenté par la figure 2.10,  $Transiter(\{1, 5\}, a) = \{3, 8, 6\}$ ,  $Transiter(\{1\}, b) = \{7\}$ ,  $Transiter(\{2\}, b) = \{ \}$

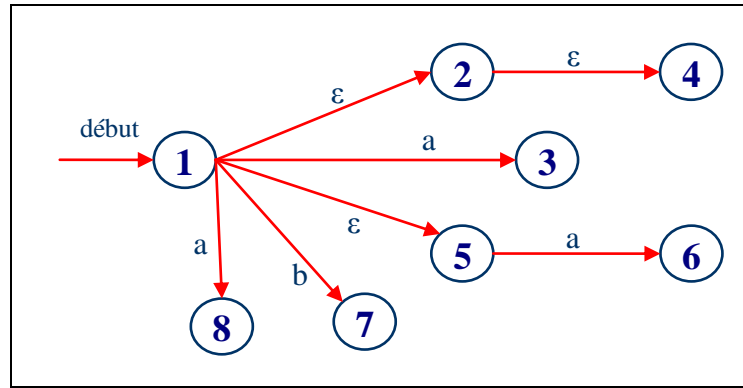


Figure 2.10. Exemple d'AFN pour le calcul de *Transiter*

### b- Principe de l'algorithme de construction de sous ensembles

Chaque état de D correspond à un sous ensemble d'états de N de sorte que D simule en parallèle tous les déplacements possibles que N peut effectuer pour une chaîne d'entrée donnée. Ceci signifie que chaque état de D correspond à un ensemble d'états de N dans lesquels N pourrait se trouver après avoir lu une suite donnée de symboles d'entrée, en incluant toutes les  $\epsilon$ -transitions possibles avant ou après la lecture des symboles.

L'état de départ de D est  $\epsilon$ -fermeture de  $e_0$ . Un état de D est final s'il correspond à un ensemble d'états de N contenant, au moins, un état final. On ajoute des états et des transitions à D en utilisant l'algorithme de construction de sous ensembles qui se présente comme suit :

```

Algorithme ConstructionSousEnsembles ;
Début
   $\epsilon$ -fermeture( $e_0$ ) est l'unique état de Détats et il est non marqué;
  Tantque il existe un état non marqué dans Détats Faire
    Début
      marquer T ;
      Pour chaque symbole d'entrée a faire
        Début
           $U \leftarrow \epsilon$ -fermeture(Transiter(T,a)) ;
          Si  $U \notin$  Détats Alors Ajouter U comme nœud non marqué à Détats
          DTrans[T, a]  $\leftarrow$  U ;
        Fin ;
      Fin ;
    Fin ;

```

**Exemple :** Application de l'algorithme de construction de sous ensemble sur l'AFN obtenu dans la figure 2.6.

$$\epsilon\text{-fermeture}(0) = \{0, 1, 2, 4, 7\} = A$$

Pour T=A

Marquer A

$$\epsilon\text{-fermeture}(\text{Transiter}(A, a)) = \epsilon\text{-fermeture}(\{3, 8\}) = \{3, 6, 1, 2, 4, 7, 8\} = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$\mathbf{DTran[A, a] \leftarrow B}$$

$$\epsilon\text{-fermeture}(\text{Transiter}(A, b)) = \epsilon\text{-fermeture}(\{5\}) = \{5, 6, 1, 2, 4, 7\} = \{1, 2, 4, 5, 6, 7\} = C$$

$$\mathbf{DTran[A, b] \leftarrow C}$$

On continue à appliquer l'algorithme avec les états actuellement non marqués B et C.

Pour T=B

Marquer B

$$\epsilon\text{-fermeture}(\text{Transiter}(B, a)) = \epsilon\text{-fermeture}(\{3, 8\}) = B$$

$$\mathbf{DTran[B, a] \leftarrow B}$$

$$\epsilon\text{-fermeture}(\text{Transiter}(B, b)) = \epsilon\text{-fermeture}(\{5, 9\}) = \{5, 6, 1, 2, 4, 7, 9\} = \{1, 2, 4, 5, 6, 7, 9\} = D$$

**DTran [B, b] ← D**

Pour T=C

Marquer C

$\varepsilon\text{-fermeture}(\text{Transiter}(C, a)) = \varepsilon\text{-fermeture}(\{3, 8\}) = B$

**DTran [C, a] ← B**

$\varepsilon\text{-fermeture}(\text{Transiter}(C, b)) = \varepsilon\text{-fermeture}(\{5\}) = \{5, 6, 1, 2, 4, 7\} = \{1, 2, 4, 5, 6, 7\} = C$

**DTran [C, b] ← C**

On continue à appliquer l'algorithme avec le seul état actuellement non marqué D.

Pour T=D

Marquer D

$\varepsilon\text{-fermeture}(\text{Transiter}(D, a)) = \varepsilon\text{-fermeture}(\{3, 8\}) = B$

**DTran [D, a] ← B**

$\varepsilon\text{-fermeture}(\text{Transiter}(D, b)) = \varepsilon\text{-fermeture}(\{5, 10\}) = \{5, 6, 1, 2, 4, 7, 10\} = \{1, 2, 4, 5, 6, 7, 10\} = E$

**DTran [D, b] ← E**

On continue à appliquer l'algorithme avec le seul état actuellement non marqué E.

Pour T=E

Marquer E

$\varepsilon\text{-fermeture}(\text{Transiter}(E, a)) = \varepsilon\text{-fermeture}(\{3, 8\}) = B$

**DTran [E, a] ← B**

$\varepsilon\text{-fermeture}(\text{Transiter}(E, b)) = \varepsilon\text{-fermeture}(\{5\}) = C$

**DTran [E, b] ← C**

Après plusieurs itérations, on arrive ainsi à un point où tous les sous ensembles d'états de l'AFD sont marqués. A la fin les cinq ensembles d'états construits sont :

- $A = \{0, 1, 2, 4, 7\}$
- $B = \{1, 2, 3, 4, 6, 7, 8\}$
- $C = \{1, 2, 4, 5, 6, 7\}$
- $D = \{1, 2, 4, 5, 6, 7, 9\}$
- $E = \{1, 2, 4, 5, 6, 7, 10\}$

A est l'état initial de l'automate et E son état final.

La table DTrans de l'AFD obtenu après l'application de l'algorithme de sous-ensembles à l'AFN de la figure 2.6., est donnée par :

Etats	Symboles	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

La figure 2.11 donne le graphe de transition de l'AFD obtenu :

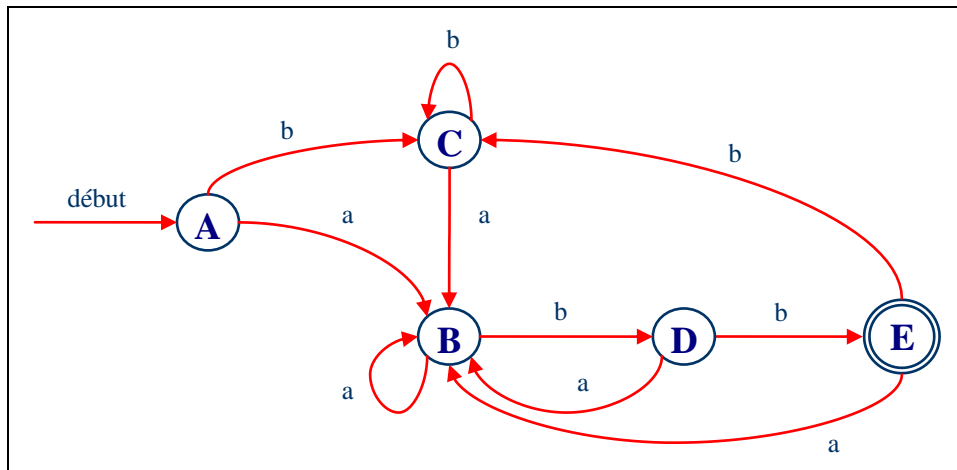


Figure 2.11. AFD équivalent à l'AFN de la figure 2.6

### c- Autre formulation de l'algorithme de construction de sous ensembles

L'algorithme de construction de sous-ensembles peut être formulé d'une manière plus condensée en utilisant une représentation tabulaire. Notons que le principe est toujours le même que dans la première formulation présentée précédemment.

- 1- Commencer avec la  $\varepsilon$ -fermeture de l'état initial (elle représente le nouvel état initial) ;
- 2- Rajouter dans la table de transition toutes les  $\varepsilon$ -fermetures des nouveaux états produits avec leurs transitions ;
- 3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel état ;
- 4- Tous les  $\varepsilon$ -fermetures contenant au moins un état final du premier automate deviennent finaux ;
- 5- Renommer les états en tant qu'états simples.

**Exemple :** Application de cette formulation de l'algorithme de construction de sous ensemble sur l'AFN obtenu dans la figure 2.6.

La table de transition de l'AFD obtenu est donnée par :

Etats	Symboles	
	a	b
0,1,2,4,7	1,2,3,4,6,7,8	1,2,4,5,6, 7
1,2,3,4,6,7,8	1,2,3,4,6,7,8	1,2,4,5,6,7,9
1,2,4,5,6, 7	1,2,3,4,6,7,8	1,2,4,5,6, 7
1,2,4,5,6,7,9	1,2,3,4,6,7,8	1,2,4,5,6,7,10
1,2,4,5,6,7,10	1,2,3,4,6,7,8	1,2,4,5,6, 7

Après renumérotation des états, nous obtenons la table suivante qui est identique à celle obtenue avec la première formulation de l'algorithme de construction de sous ensembles (voir juste avant la figure 2.11) :

Etats	Symboles	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

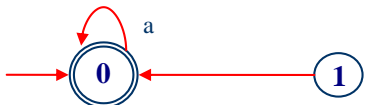
### 2.5.5 Minimisation d'un AFD (étape 5)

Intuitivement, la minimisation d'un AFD consiste en deux opérations principales :

- Elimination de tous les états inaccessibles (et improductifs) de l'AFD
- Regroupement des états équivalents en les remplaçant par des classes d'équivalence d'états

Sachant que :

- Un état  $e$  est accessible s'il existe une chaîne menant à  $e$  à partir de l'état initial. Un état, autre que l'état initial, est inaccessible lorsqu'aucun arc n'arrive sur cet état dans un chemin provenant de l'état initial.

Par exemple, dans l'automate  l'état 1 est inaccessible.

- On dit que deux états sont équivalents si ces états permettent d'atteindre un état final à travers la même chaîne. En d'autres termes, tous les suffixes reconnus à partir de ses états sont exactement les mêmes.

Par exemple, la figure 2.12 montre la minimisation d'un automate contenant deux états équivalents (états 3 et 4).

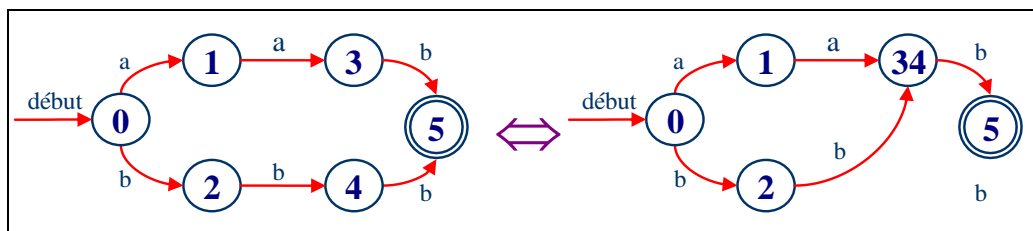


Figure 2.12. Exemple de minimisation d'un AFD

Un algorithme simplifié de minimisation peut être utilisé, il fonctionne par raffinements successifs en définissant des classes d'équivalence d'états qui vont correspondre aux états du nouvel automate (minimisé). Cet algorithme comprend les étapes suivantes :

1. Définir deux classes, C1 contenant les états finaux et C2 les états non finaux.
2. S'il existe un symbole  $a$  et deux états  $e_1$  et  $e_2$  d'une même classe tels que  $\text{Transiter}(e_1, a)$  et  $\text{Transiter}(e_2, a)$  n'appartiennent pas à la même classe (d'arrivée), alors créer une nouvelle classe et séparer  $e_1$  et  $e_2$ . On laisse dans la même classe tous les états qui donnent un état d'arrivée dans la même classe.
3. Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
4. Chaque classe restante forme un état du nouvel automate.

**Exemple** (traité pendant les séances de cours)

Effectuer la minimisation des automates suivants :

- 1). L'AFD de la figure 2.11 (page 21).
- 2). L'AFD obtenu dans la section 2.5.4.1 page 16 (états de A à M).

### 2.5.6 Simulation d'un AFD (étape 6)

Soit une chaîne  $x$  représentée par un fichier de caractères se terminant par EoF qui est un caractère spécial indiquant la fin d'un fichier. Soit un AFD  $D$  avec un état initial  $e_0$  et un ensemble d'état finaux  $F$ .

**carsuiv** est une procédure qui retourne le caractère suivant du texte source dans  $ch$  (passé en paramètre). La fonction **Transiter**( $e, ch$ ) donne l'état vers lequel il y a transition à partir de l'état  $e$  avec le caractère  $ch$ .

L'algorithme de simulation de l'AFD est le suivant :

```

Algorithme SimulAFD ;
Début
  e ← e0 ;
  carsuiv(ch) ;
  Tantque ch ≠ EoF Faire
    Début
      e ← Transiter(e, ch) ;
      carsuiv(ch) ;
    Fin ;
  Si e ∈ F Alors accepter(x)
  Sinon rejeter(x) ;
Fin ;

```

## 2.7 Générateurs d'analyseurs lexicaux

Les étapes de construction d'un analyseur lexical à partir de la spécification des unités lexicales peuvent être automatisées en utilisant un outil logiciel dit générateur d'analyseur lexical tel que l'outil LEX qui peut être considéré comme le plus ancien. Cette approche de construction des analyseurs lexicaux est considérée comme la plus simple car on se limite à fournir les définitions exactes des unités lexicales du langage considéré et le générateur fait le reste en générant le code source de l'analyseur lexical.

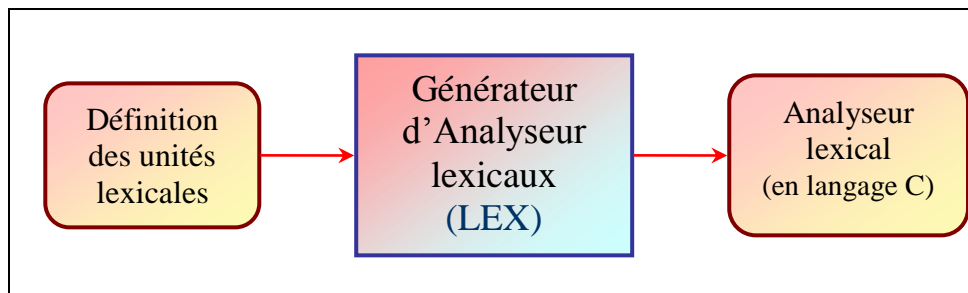


Figure 2.13. Construction d'un analyseur lexical en utilisant un générateur

Le générateur LEX permet de générer un analyseur lexical en langage C à partir des spécifications des unités lexicales exprimées en langage LEX. Le noyau de LEX permet de produire l'automate d'état fini correspondant à la reconnaissance de l'ensemble des unités lexicales spécifiées. Ainsi, LEX construit une table de transition pour un automate à états finis à partir des modèles d'expressions régulières de la spécification en langage LEX.

L'analyseur lexical lui-même consiste en un simulateur d'automate à états finis (en langage C), qui utilise la table de transition obtenue pour rechercher les unités lexicales dans le texte en entrée. La spécification des unités lexicales est effectuée en utilisant le langage LEX, en créant un programme */ex/* constitué de trois parties :

```

Déclarations
%%
Règles de traduction
%%
Procédures auxiliaires

```

La première partie contient des déclarations de variables, de constantes et des définitions régulières utilisées comme composantes des expressions régulières qui apparaissent dans les règles de traduction.

Les règles de traduction sont des instructions (en langage C) de la forme  $m_i \{action_i\}$  où chaque  $m_i$  est une expression régulière et chaque  $action_i$  est un fragment de programme qui décrit quelle action l'analyseur lexical devrait réaliser quand une unité lexicale concorde avec le modèle  $m_i$ .

Les procédures auxiliaires sont toutes les procédures qui pourraient être utiles dans les actions.

## **SERIE DE TD N°:1 COMPILATION**

### **ANALYSE LEXICALE**

#### **Exercice 1**

Proposer des définitions régulières et des automates à états finis correspondant aux unités lexicales suivantes :

1. des nombres entiers ou réels non signés tels que 2013, 12.33, 314.3E-2, 0.314E+1, 0.314E2, 314E-2 dans lesquels la partie décimale, si elle existe, commence par le point décimal, qui doit être suivi par un chiffre au minimum. La partie exposant, si elle existe, commence par un E, qui est suivi ou non par un signe (+ ou -), puis un chiffre au minimum. Le nombre de chiffres dans les parties entière, décimale et exposant n'est pas limité.
2. des identificateurs qui ne dépassent pas 4 caractères (lettres ou chiffres) dont le premier est obligatoirement une lettre.
3. des nombres entiers ou réels non signés écrits en utilisant une notation scientifique sans point décimal. La partie exposant, si elle existe, commence par un E qui est obligatoirement suivi par un signe (+ ou -), puis un chiffre au minimum. Le nombre de chiffres est limité à 3 pour les entiers. Pour les réels, il y a au maximum 3 chiffres avant le E et 2 chiffres après. Exemples : 257, 36E+02, 245E-3
4. des noms de procédures et de fonctions qui commencent obligatoirement par les 4 lettres **P R O C** pour les procédures et **F O N C** pour les fonctions. Ces 4 premières lettres peuvent être en majuscules ou en minuscules et sont suivies par 7 autres caractères au maximum dont les 4 premiers sont des lettres (au minimum deux lettres, la première étant obligatoirement une majuscule) et les 3 restants sont des chiffres (les chiffres ne sont pas obligatoires). Exemples : *ProcTest10, foncMin, FONCFc2, PROCPr36, procRech*

#### **Exercice 2**

Soit un automate d'états finis dont l'ensemble des états est  $\{0, 1, 2, 3, 4\}$ , état initial: 0, états finaux: 0 et 3, le vocabulaire étant  $\{a, b\}$ . Notons T la fonction de transition de cet automate.

$T(0, a) = \{2\}$ ,  $T(0, b) = \{1\}$ ,  $T(1, a) = \{2\}$ ,  $T(1, b) = \{1\}$ ,  $T(2, a) = \{3\}$ ,  $T(2, b) = \{2\}$ ,

$T(3, a) = \{2, 4\}$ ,  $T(3, b) = \{1, 3\}$ ,  $T(4, a) = \{4\}$ ,  $T(4, b) = \{4\}$

1. Etablir la table de transition et la représentation graphique de l'automate.
2. Etablir la table de transition et la représentation graphique de l'automate déterministe équivalent.
3. Donner le chemin de reconnaissance des chaînes *bababb* et *abb* par l'automate déterministe obtenu.
4. Donner le chemin de reconnaissance de la chaîne *aaab* par l'automate initial puis proposer une minimisation de cet automate qui n'affecte pas le langage qu'il reconnaît.

#### **Exercice 3**

Soit un langage, représentant un petit sous-ensemble du langage Pascal, et constitué par les unités lexicales suivantes :

- Des identificateurs commençant par une lettre suivie d'une combinaison quelconque de lettres ou de chiffres
- Des constantes numériques entières non signées (sans limitation de longueur)
- L'affectation ( := )

Construire l'analyseur lexical correspondant à ce langage en utilisant l'approche basée sur les automates finis.

#### **Exercice 4**

Soient les expressions régulières suivantes: *ab*, *ab<sup>+</sup>* et *(a|b)<sup>\*</sup>*

Appliquer les étapes 2 à 5 de la démarche de construction d'un analyseur lexical basée sur les automates d'états finis, afin de construire un analyseur lexical pour l'ensemble des trois expressions régulières précédentes, correspondant à des unités lexicales.

#### **Exercice 5**

Soient les expressions régulières suivantes: *a*, *abb*, *a<sup>\*</sup>b<sup>+</sup>*

1). Appliquer les étapes 2 à 5 de la démarche de construction d'un analyseur lexical basée sur les automates d'états finis, afin de construire un analyseur lexical pour l'ensemble des trois expressions régulières précédentes, correspondant à des unités lexicales.

2). Décrire l'analyse lexicale des chaînes *aaba* et *abb* en donnant leur chemin de reconnaissance (utiliser l'algorithme de simulation d'un AFD qui représente la 6<sup>ème</sup> étape de construction de l'analyseur).



## **CHAPITRE 3 : ANALYSE SYNTAXIQUE : INTRODUCTION, RAPPELS ET COMPLEMENTS**

---

### **3.1 Rôle d'un analyseur syntaxique**

---

L'analyseur syntaxique (Parser en anglais) a comme rôle l'analyse des séquences d'unités lexicales, obtenues par l'analyseur lexical, conformément à une grammaire qui engendre le langage considéré.

L'analyseur syntaxique reconnaît la structure syntaxique d'un programme source et produit une représentation qui est généralement sous forme d'un arbre syntaxique. Ce dernier peut être décoré par des informations sémantiques puis utilisé pour produire un programme cible.

Pour les langages simples, l'analyseur syntaxique peut ne pas livrer une représentation explicite de la structure syntaxique du programme, mais remplit le rôle d'un module principal qui appelle des procédures d'analyse lexicale, d'analyse sémantique et de génération de code.

Ainsi, les principales fonctions d'un analyseur syntaxique peuvent être résumées comme suit :

1. L'analyse de la chaîne des unités lexicales délivrées par l'analyseur lexical, pour vérifier si cette chaîne peut être engendrée par la grammaire du langage considéré.
2. La détection des erreurs syntaxiques si la syntaxe du langage n'est pas respectée.
3. La construction éventuelle d'une représentation interne qui sera utilisée par les phases ultérieures.

### **3.2 Approches d'analyse syntaxique**

---

Il existe trois grandes catégories de méthodes pour l'analyse syntaxique :

1. Méthodes universelles : Elles sont généralement tabulaires comme celles de Cocke-Younger-Kasami (1965-1967) et de Earley (1970) qui peuvent analyser une grammaire quelconque. Cependant, ces méthodes sont trop inefficaces pour être utilisées dans les compilateurs industriels.
2. Méthodes descendantes : Ces méthodes sont dites aussi **Top-Down** car elles construisent des arbres d'analyse de haut en bas (de la racine aux feuilles).
3. Méthodes ascendantes : Ces méthodes sont dites aussi **Bottom-up** car elles construisent des arbres d'analyse de bas en haut (remontent des feuilles à la racine).

#### **Exemple d'analyse**

Soit la grammaire  $G = \{V_T, V_N, S, P\}$

P étant défini par les règles suivantes :

$$\begin{aligned} S &\rightarrow B \\ B &\rightarrow R \mid (B) \\ R &\rightarrow E = E \\ E &\rightarrow a \mid b \mid (E+E) \end{aligned}$$

$V_T = \{a, b, (, ), =, +\}$

$V_N = \{S, B, R, E\}$

L'analyse de la chaîne ( $a = (b + a)$ ) par une approche descendante débute par l'axiome et s'effectue par dérivations successives comme suit (si on dérive à partir de la gauche) :

$$\begin{aligned} S &\rightarrow B \\ &\rightarrow ( B ) \\ &\rightarrow ( R ) \\ &\rightarrow ( E = E ) \\ &\rightarrow ( a = E ) \\ &\rightarrow ( a = ( E + E ) ) \\ &\rightarrow ( a = ( b + E ) ) \\ &\rightarrow ( a = ( b + a ) ) \end{aligned}$$

L'analyse de la chaîne ( $a = (b + a)$ ) par une approche ascendante débute au niveau de la chaîne elle-même (**lire de bas en haut**) et remplace, à chaque étape, une partie de la chaîne par un non-terminal, par réductions successives, comme suit (si on réduit à partir de la gauche) :

$$\begin{aligned} S \\ B \\ (B) \\ ( R ) \\ ( E = E ) \\ ( E = ( E + E ) ) \\ ( E = ( E + a ) ) \\ ( E = ( b + a ) ) \\ ( a = ( b + a ) ) \end{aligned}$$

Les méthodes ascendantes et descendantes sont utilisées fréquemment pour la mise en oeuvre de compilateurs. Dans les deux cas, l'entrée de l'analyseur syntaxique est parcourue de la gauche vers la droite, un symbole à la fois. Les méthodes ascendantes et descendantes les plus efficaces fonctionnent uniquement avec des sous-classes de grammaires telles que les grammaires LL et LR (qui seront étudiées aux chapitres 4 et 5) qui sont suffisamment expressives pour décrire la majorité des structures syntaxiques des langages de programmation.

Les analyseurs syntaxiques mis en oeuvre manuellement utilisent généralement des méthodes descendantes et des grammaires LL. C'est le cas, par exemple, pour l'analyse prédictive et la descente réursive.

Les méthodes ascendantes sont souvent utilisées par les outils de construction des analyseurs syntaxiques. C'est le cas avec les analyseurs LR qui sont des analyseurs plus généraux que les analyseurs descendants.

La construction d'analyseurs LR est courante dans les environnements de développement de compilateurs. Notons qu'il existe aussi d'autres méthodes ascendantes telles que la précedence d'opérateurs, la précedence simple, la précedence faible et les matrices de transition.

## ***3.3 Concepts de base***

---

### ***3.3.1 Notion de grammaire***

---

Une grammaire  $G$  est un formalisme qui permet la génération des chaînes d'un langage. Elle est formellement définie par un quadruplet  $G = (V_T, V_N, S, P)$  où

- $V_T$  est le vocabulaire terminal contenant l'ensemble des symboles atomiques individuels, pouvant être composés en séquence pour former des phrases et souvent notés en lettres minuscules.

- $V_N$  est le vocabulaire non terminal contenant l'ensemble des symboles non terminaux représentant des constructions syntaxiques. Ces symboles sont souvent notés en majuscules.
- $S$  est appelé axiome ou symbole initial de la grammaire.
- $P$  est l'ensemble des règles de production de la forme  $\alpha \rightarrow \beta$  où  $\alpha$  et  $\beta$  appartiennent à  $(V_T \cup V_N)^*$

Selon la forme des règles de production, il existe quatre types de grammaires que nous présentons selon la classification de Chomsky. Dans la suite nous considérons que si  $V$  est un vocabulaire alors  $V^*$  est l'ensemble de toutes les chaînes résultant de la concaténation des éléments de  $V$ .

Type de grammaire	Forme des règles de production
Type 0 (Grammaires sans restrictions)	$\alpha \rightarrow \beta$ avec $\alpha \in (V_T \cup V_N)^+$ et $\beta \in (V_T \cup V_N)^*$ Autrement dit il n'y a aucune restriction sur $\alpha$ et $\beta$
Type 1 (Grammaires à contexte lié ou Contextuelles)	$\alpha \rightarrow \beta$ avec $\alpha \in (V_T \cup V_N)^+$ et $\beta \in (V_T \cup V_N)^*$ et $ \alpha  \leq  \beta $ et $\epsilon$ (la chaîne vide) ne peut être généré que par l'axiome avec la règle $S \rightarrow \epsilon$
Type 2 (Grammaires à contexte libre, non contextuelles, algébriques ou encore de Chomsky)	$A \rightarrow \alpha$ avec $A \in V_N$ et $\alpha \in (V_T \cup V_N)^*$
Type 3 (Grammaires régulières ou linéaires droites)	$A \rightarrow \alpha B$ ou $A \rightarrow \alpha$ avec $A, B \in V_N$ et $\alpha \in V_T^*$

### 3.3.2 Expression des grammaires

A part l'énumération des règles de production, il est possible de définir une grammaire en utilisant des notations textuelles telles que la **forme BNF** ou la **forme EBNF** ainsi que des représentations graphiques telles que les **diagrammes syntaxiques**.

#### a- La notation BNF (Backus-Naur Form)

Cette notation est due aux créateurs de Fortran (J. Backus, 1955) et Algol (P. Naur, 1963). Les règles sont écrites avec les conventions suivantes :

- Un symbole  $A \in V_N$  est noté par  $\langle A \rangle$  ou  $A$  (en utilise des lettres majuscules)
- Un symbole  $a \in V_T$  est noté sans les  $\langle \rangle$  ou encore par " $a$ " ou  $a$  (en utilise des lettres minuscules)
- Le signe  $\rightarrow$  devient  $::=$
- Un ensemble de règles  $A \rightarrow \alpha, A \rightarrow \beta, \dots, A \rightarrow \gamma$  sera remplacé par  $A \rightarrow \alpha | \beta | \dots | \gamma$

#### Exemple

$\langle \text{Bloc} \rangle ::= \text{Begin } \langle \text{List Opt Inst} \rangle \text{ End } .$

$\langle \text{List Opt Inst} \rangle ::= \langle \text{List Inst} \rangle | \epsilon$

$\langle \text{List Inst} \rangle ::= \langle \text{List Inst} \rangle ; \langle \text{Inst} \rangle | \langle \text{Inst} \rangle$

$\langle \text{Inst} \rangle ::= \text{id} := \langle \text{Exp} \rangle | \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Inst} \rangle | \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Inst} \rangle \text{ else } \langle \text{Inst} \rangle$

#### b- La notation EBNF (Extended Backus-Naur Form)

C'est une représentation plus réduite que BNF qui permet d'écrire les grammaires sous une forme plus condensée en utilisant les signes  $\{ \}$  et  $[ ]$ .

- Une partie optionnelle est notée entre  $[ ]$  (apparaît 0 ou 1 fois)
- Une partie qui peut apparaître de façon répétée est notée entre  $\{ \}$  (apparaît 0 ou N fois)

Notons que, dans les notations BNF et EBNF, les parenthèses peuvent être utilisées pour éviter les ambiguïtés, comme c'est le cas dans la règle suivante :

$\langle \text{Inst} \rangle ::= \text{For } \text{id} := \langle \text{Exp} \rangle ( \text{to} \mid \text{downto} ) \langle \text{Exp} \rangle \text{ do } \langle \text{Bloc} \rangle$

la partie ( **to** | **downto** ) exprime un choix entre **to** et **downto**

### Exemple

L'exemple présenté avec la notation BNF peut être noté comme suit en utilisant EBNF :

$\langle \text{Bloc} \rangle ::= \text{Begin } [ \langle \text{List Inst} \rangle ] \text{ End } .$

$\langle \text{List Inst} \rangle ::= \langle \text{Inst} \rangle \{ ; \langle \text{Inst} \rangle \}$

$\langle \text{Inst} \rangle ::= \text{id} := \langle \text{Exp} \rangle \mid \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Inst} \rangle [ \text{else } \langle \text{Inst} \rangle ]$

Cette représentation est la même que la suivante :

$\langle \text{Bloc} \rangle ::= \text{Begin } [ \langle \text{Inst} \rangle \{ ; \langle \text{Inst} \rangle \} ] \text{ End } .$

$\langle \text{Inst} \rangle ::= \text{id} := \langle \text{Exp} \rangle \mid \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Inst} \rangle [ \text{else } \langle \text{Inst} \rangle ]$

## c- Les diagrammes syntaxiques

C'est une représentation sous forme graphique des règles du langage où les terminaux sont **encerclés** et les non terminaux sont **encadrés**. La figure 3.1 donne les diagrammes syntaxiques correspondant à l'exemple précédent.

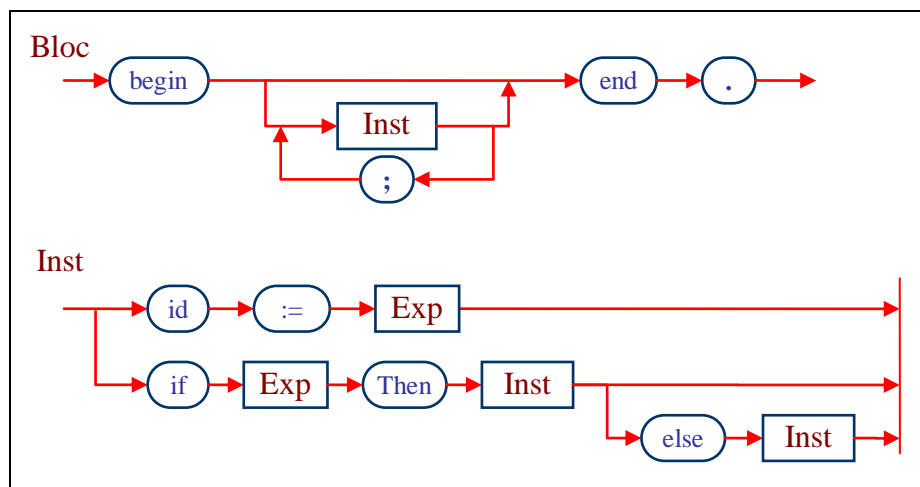


Figure 3.1. Exemple de diagrammes syntaxiques

### 3.3.3 Dérivation et arbres syntaxiques

#### a- Dérivation

On appelle dérivation, l'application d'une ou plusieurs règles à partir d'une phrase (chaîne) de  $(V_T \cup V_N)^+$ .

On note  $\rightarrow$  une dérivation obtenue par l'application d'une seule règle de production et on <sup>\*</sup>note  $\rightarrow^*$  une dérivation obtenue par l'application de plusieurs règles de production.

#### Exemple

Soit les règles :

$S \rightarrow A$

$A \rightarrow bB$

$$B \rightarrow c$$

La chaîne bc peut être dérivée par l'application de ces règles comme suit :

$$\begin{aligned} S &\rightarrow A \\ &\rightarrow bB \\ &\rightarrow bc \end{aligned}$$

On peut noter ces dérivations par une expression plus condensée  $S \xrightarrow{*} bc$

D'une manière générale, si  $G$  est une grammaire définie par  $(V_T, V_N, S, P)$  alors le langage généré peut être noté par :  $L(G) = \{x \in V_T^* \text{ tel que } S \rightarrow x\}$ .

On distingue deux façons d'effectuer une dérivation :

- Dérivation gauche (la plus à gauche ou « left most ») : Elle consiste à remplacer toujours le symbole non terminal le plus à gauche.
- Dérivation droite (la plus à droite ou « right most ») : Elle consiste à remplacer toujours le symbole non terminal le plus à droite.

### Exemple

Soit  $G = (\{a, +, (, )\}, \{E, F\}, E, P)$

$P$  est défini par :

$$\begin{aligned} E &\rightarrow E + F \\ E &\rightarrow F \\ F &\rightarrow (F) \mid a \end{aligned}$$

Peut-on affirmer que  $a+a$  appartient au langage  $L(G)$  ?

Par dérivations gauches, on a :

$$E \rightarrow E + F \rightarrow F + F \rightarrow a + F \rightarrow a + a \quad \text{donc} \quad E \xrightarrow{*} a+a \quad \text{et la chaîne appartient au langage}$$

Par dérivations droites, on a :

$$E \rightarrow E + F \rightarrow E + a \rightarrow F + a \rightarrow a + a \quad \text{donc} \quad E \xrightarrow{*} a+a \quad \text{et la chaîne appartient au langage}$$

### b- Arbre de dérivation (Arbre syntaxique)

On appelle **arbre de dérivation** ou **arbre syntaxique concret** un arbre tel que

1. La racine est l'axiome de la grammaire
2. Les feuilles de l'arbre sont des symboles terminaux
3. Les nœuds sont les symboles non terminaux
4. Pour un nœud interne d'étiquette  $A$ , le mot  $\alpha_1\alpha_2 \dots \alpha_n$  obtenu en lisant de gauche à droite les étiquettes des nœuds fils de  $A$  correspond à une règle de la grammaire  $A \rightarrow \alpha_1\alpha_2 \dots \alpha_n$
5. Le mot  $m$ , dont on fait l'analyse, est constitué des étiquettes des feuilles, lues de gauche à droite

### Exemple

Soit  $G = (\{a, b, c\}, \{S, T\}, S, P)$

$P$  est défini par :

$$\begin{aligned} S &\rightarrow aTb \mid c \\ T &\rightarrow cS \mid S \end{aligned}$$

L'arbre de dérivation de  $accacbb$  est donné par la figure 3.2.

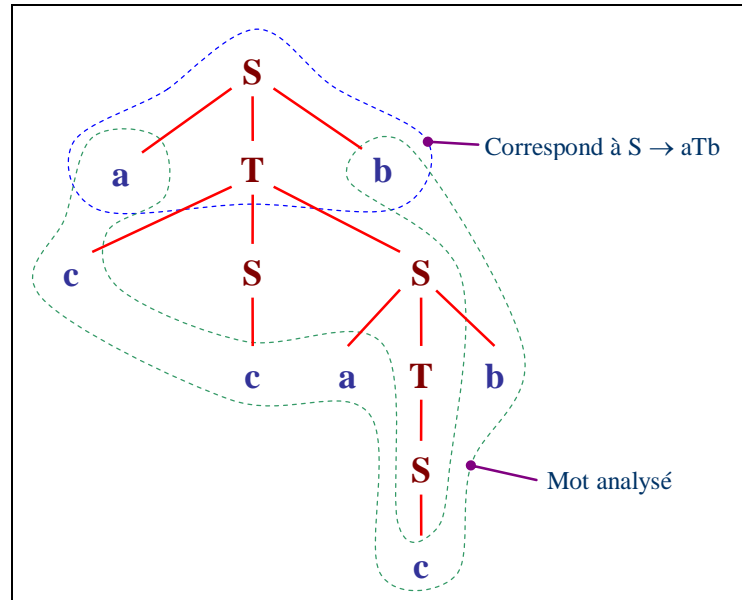


Figure 3.2. Exemple d'arbre de dérivation

### c- Arbre abstrait

On appelle **arbre syntaxique abstrait** ou **arbre abstrait** une forme condensée d'arbre syntaxique qui est utilisée dans les compilateurs. L'arbre abstrait est obtenu par des transformations simples de l'arbre de dérivation.

L'arbre syntaxique abstrait est plus compact que l'arbre syntaxique concret et contient des informations sur la suite des actions effectuées par un programme. Dans un arbre abstrait, les opérateurs et les mots clés apparaissent comme des nœuds intérieurs et les opérandes apparaissent comme des feuilles.

#### Exemple

Si on considère, la grammaire  $G = (\{a, +, (, )\}, \{E, T, F\}, E, P)$

P étant défini par :

$E \rightarrow T \mid E + T$

$T \rightarrow F \mid T * F$

$F \rightarrow a \mid (E)$

Les arbres syntaxiques (concret et abstrait) pour la chaîne  $a + a * a$  sont donnés par la figure 3.5.

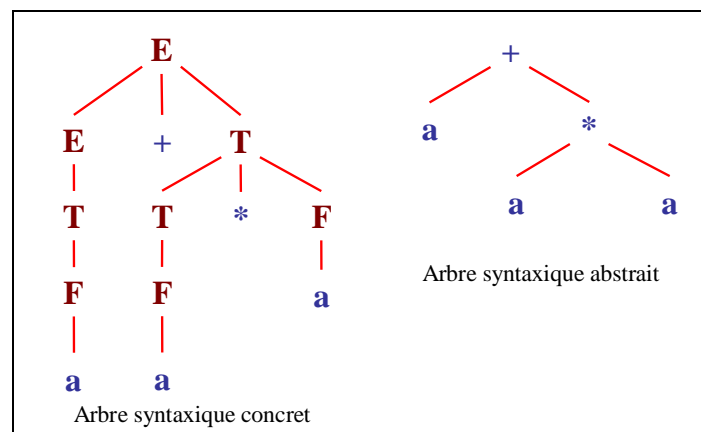


Figure 3.4. Arbre syntaxique concret et abstrait pour la chaîne  $a + a * a$

### d- Grammaire ambiguë

On dit qu'une grammaire est ambiguë si elle produit plus d'un arbre syntaxique pour une chaîne donnée.

Si on considère la grammaire  $G = (\{+, -, *, /, id\}, \{E\}, E, P)$  où P est défini par :

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid id$$

La figure 3.3 donne deux arbres de dérivation pour la chaîne  $id + id * id$

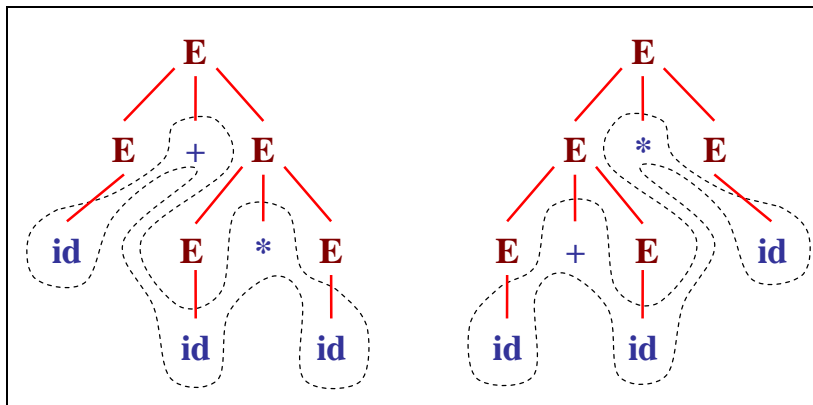


Figure 3.3. Deux arbres de dérivation pour la chaîne  $id + id * id$

L'élimination de l'ambiguïté est une tâche importante qu'on exige généralement pour les langages de programmation qui sont basés sur des grammaires non contextuelles. L'élimination de l'ambiguïté peut avoir lieu de deux façons :

1. Élimination de l'ambiguïté par ajout de règles. Dans cette approche, la grammaire reste inchangée, on ajoute, à la sortie de l'analyseur syntaxique, quelques règles dites de désambiguïté, qui éliminent les arbres syntaxiques non désirables et conservent un arbre unique pour chaque chaîne. On peut, par exemple, éliminer l'ambiguïté en affectant des priorités aux opérateurs arithmétiques. Dans les expressions arithmétiques, la priorité est affectée aux parenthèses, puis au signe moins unaire, puis à la multiplication/division et enfin à l'addition/soustraction.

2. Réécriture de la grammaire. Il s'agit de changer la grammaire en incorporant des règles de priorité et d'associativité. Autrement dit, on ajoute de nouvelles règles et de nouveaux symboles non terminaux. La grammaire ambiguë donnée précédemment peut être désambiguïsée comme suit :

$G = (\{+, -, *, /, id\}, \{E, T, F, L\}, E, P)$  où  $P$  est défini par :

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow -F \mid L$$

$$L \rightarrow (E) \mid id$$

La chaîne  $id + id * id$  a maintenant une seule séquence de dérivations (gauches) :

$$E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow L + T \rightarrow id + T \rightarrow id + T * F$$

$$\rightarrow id + F * F \rightarrow id + L * F \rightarrow id + id * F \rightarrow id + id * L \rightarrow id + id * id$$

et l'unique arbre syntaxique donné par la figure 3.4.

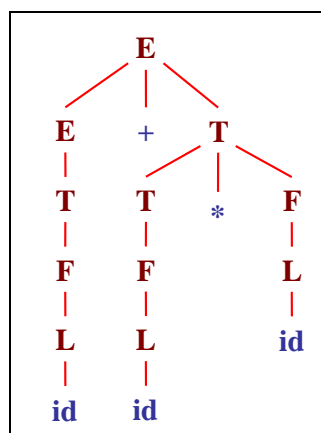


Figure 3.4. Arbre syntaxique de  $id + id * id$

## **SERIE DE TD N°:2 COMPILATION**

### **ANALYSE SYNTAXIQUE : INTRODUCTION, RAPPELS ET COMPLEMENTS**

#### **Exercice 1**

1) Soit la grammaire  $G = (\{a, b\}, \{S, A, B, C\}, S, P)$  où  $P$  est défini par :

$S \rightarrow AB \mid BC$

$A \rightarrow BA \mid a$

$B \rightarrow CC \mid b$

$C \rightarrow AB \mid a$

Analyser la chaîne **babaab** de manière descendante puis ascendante, en construisant, à chaque fois, son arbre de dérivation.

#### **Exercice 2**

1) Soit la grammaire des expressions arithmétiques  $G = (\{+, -, *, /, a, b, c, (, )\}, \{E\}, E, P)$  où  $P$  est défini par :

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid a \mid b \mid c$

Donner l'arbre de dérivation de la chaîne **b + a \* b - c** ? Que peut-on en déduire ?

2) Soit la grammaire  $G' = (\{+, -, *, /, a, b, (, )\}, \{E, T, F, L\}, E, P)$  où  $P$  est défini par :

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow -F \mid L$

$L \rightarrow (E) \mid a \mid b \mid c$

a) Donner l'arbre de dérivation de la chaîne **b + a \* b - c**. Que peut-on en déduire ?

b) Donner l'arbre abstrait correspondant à la chaîne **b + a \* b - c**.

c) Donner les arbres syntaxiques concret et abstrait de la chaîne **b - a + c**.

#### **Exercice 3**

1) Soit la grammaire des expressions booléennes  $G = (\{\text{ou}, \text{et}, \text{non}, \text{vrai}, \text{faux}, (, )\}, \{A\}, A, P)$  où  $P$  est défini par :

$A \rightarrow A \text{ ou } A \mid A \text{ et } A \mid \text{non } A \mid (A) \mid \text{vrai} \mid \text{faux}$

Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** ? Que peut-on en déduire ?

2) Proposer une grammaire  $G'$  pour éliminer le problème posé par la grammaire  $G$ .

3) Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** en utilisant la grammaire  $G'$ .

#### **Exercice 4**

Soit la grammaire  $G$  d'un langage proche de Pascal, exprimée sous forme EBNF de la manière suivante, exprimer la grammaire  $G$  sous forme de diagrammes syntaxiques.

$\langle \text{Programme} \rangle ::= \underline{\text{Program}} \text{ ident} ; \langle \text{Bloc} \rangle .$

$\langle \text{Bloc} \rangle ::= [ \underline{\text{Const}} \langle \text{SuitConst} \rangle ; ] [ \underline{\text{Var}} \langle \text{SuitVar} \rangle ; ] \{ \underline{\text{Procédure}} \text{ ident} ; \langle \text{Bloc} \rangle ; \}$

$\underline{\text{Begin}} \langle \text{Inst} \rangle \{ ; \langle \text{Inst} \rangle \} \underline{\text{End}}$

$\langle \text{SuitConst} \rangle ::= \langle \text{DecConst} \rangle \{ , \langle \text{DecConst} \rangle \}$

$\langle \text{DecConst} \rangle ::= \text{ident} = \text{nbEnt}$

$\langle \text{SuitVar} \rangle ::= \text{ident} \{ , \text{ident} \}$

$\langle \text{Inst} \rangle ::= \text{ident} := \langle \text{Exp} \rangle \mid \text{If } \langle \text{Cond} \rangle \text{ Then } \langle \text{Inst} \rangle [ \text{Else } \langle \text{Inst} \rangle ]$

$\mid \text{Repeat } \langle \text{Inst} \rangle \text{ Until } \langle \text{Cond} \rangle \mid \text{While } \langle \text{Cond} \rangle \text{ Do } \langle \text{Inst} \rangle \mid \underline{\text{Begin}} \langle \text{Inst} \rangle \{ ; \langle \text{Inst} \rangle \} \underline{\text{End}}$

$\langle \text{Cond} \rangle ::= \langle \text{Exp} \rangle ( = \mid < \mid > \mid < = \mid > = ) \langle \text{Exp} \rangle$

$\langle \text{Exp} \rangle ::= \langle \text{Terme} \rangle \{ ( + \mid - ) \langle \text{Terme} \rangle \}$

$\langle \text{Terme} \rangle ::= \langle \text{Facteur} \rangle \{ ( * \mid / ) \langle \text{Facteur} \rangle \}$

$\langle \text{Facteur} \rangle ::= \text{ident} \mid \text{nbEnt} \mid ( \langle \text{Exp} \rangle )$



## CHAPITRE 4 : ANALYSE SYNTAXIQUE DESCENDANTE

L'analyse descendante part de l'axiome et tente de créer une chaîne identique à la chaîne d'entrée par dérivations successives. Ainsi, l'opération de base est le remplacement d'un symbole non terminal par une de ses parties droites. Quand l'analyseur doit remplacer un non terminal ayant plus d'une partie droite, il doit pouvoir effectuer le **bon choix**. Dans ce chapitre, une méthode descendante d'analyse syntaxique va être exposée, il s'agit de la méthode d'analyse prédictive non récursive.

### 4.1 Principe de la méthode d'analyse prédictive non récursive

L'analyse prédictive non récursive est une méthode descendante déterministe où la dérivation à appliquer pour un non terminal (partie droite à choisir) est recherchée dans une table d'analyse. Un analyseur syntaxique prédictif est dirigé par une table et fonctionne en tenant à jour une pile explicite, ainsi, il possède les éléments représentés par la figure 4.1.

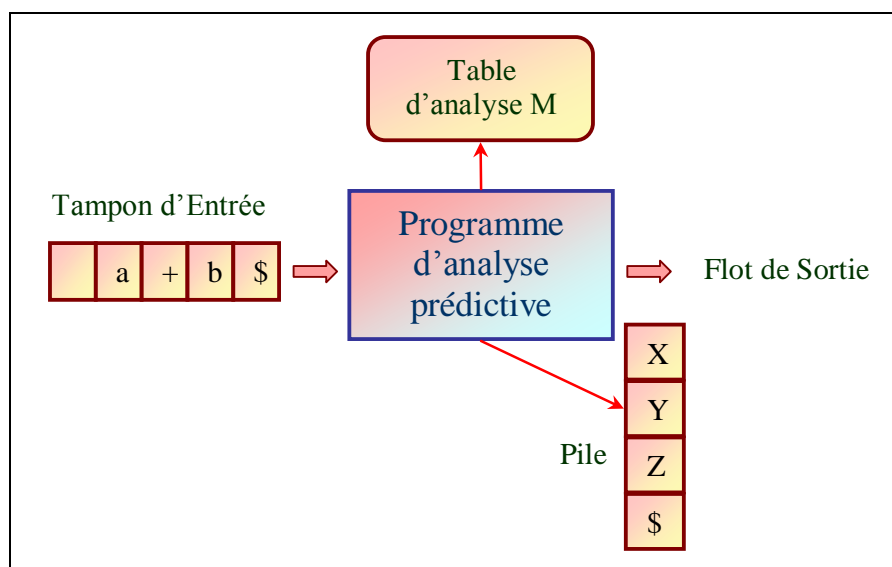


Figure 4.1. Modèle d'analyseur prédictif non récursif

- ❖ Le tampon d'entrée contient la chaîne à analyser suivie du symbole \$ qui est un marqueur de fin de chaîne.
- ❖ La pile contient une suite de symboles grammaticaux, avec \$ marquant le fond de la pile. Initialement, l'axiome S se trouve au dessus de \$.
- ❖ La table d'analyse M qui est une matrice où un élément  $M[X,a]$  correspond à un non terminal X, qu'on doit dériver, et à un terminal a (ou \$), qu'on est en train de lire. Si on est arrivé à une étape où on doit dériver le non terminal X et qu'on lit le caractère a dans la chaîne d'entrée, on doit utiliser la règle de production qui se trouve dans  $M[X, a]$ .
- ❖ Le programme d'analyse prédictive contrôle l'analyse syntaxique en considérant le symbole en sommet de pile X et le symbole d'entrée courant a. Ainsi, l'action de l'analyseur est déterminée par X et a, il y a plusieurs cas possibles (voir la section 4.4).

## 4.2 Fonctions Premier et Suivant

Les fonctions Premier et Suivant permettent d'effectuer la construction de la table M d'analyse prédictive en déterminant le contenu de ses entrées.

### 4.2.1 Fonction Premier (Début ou First)

Les symboles premiers d'une chaîne  $\alpha$  sont les terminaux (y compris  $\varepsilon$ ) pouvant se trouver au début de  $\alpha$  (commencer  $\alpha$ ) directement ou après plusieurs dérivations.

$$\text{Premier}(\alpha) = \{x \in V_T / \alpha \xrightarrow{*} x\beta, \alpha, \beta \in (V_T UV_N)^*\} \text{ sachant que : Si } \alpha \xrightarrow{*} \varepsilon \text{ alors } \varepsilon \in \text{Premier}(\alpha)$$

#### Exemple 1

Soient les règles de productions suivantes pour une grammaire donnée :

$S \rightarrow Ba$   
 $B \rightarrow cP \mid bP \mid P \mid \varepsilon$   
 $P \rightarrow dS$

On se propose de déterminer les premiers du symbole S.

On a:  $S \rightarrow Ba \rightarrow cPa$  donc:  $c \in \text{Prem}(S)$   
On a:  $S \rightarrow Ba \rightarrow bPa$  donc:  $b \in \text{Prem}(S)$   
On a:  $S \rightarrow Ba \rightarrow Pa \rightarrow dSa$  donc:  $d \in \text{Prem}(S)$   
On a:  $S \rightarrow Ba \rightarrow \varepsilon a \rightarrow a$  donc:  $a \in \text{Prem}(S)$

Nous pouvons en déduire que **Prem(S) = {a, b, c, d}**

Pour calculer Prem(X), il faut appliquer les trois règles suivantes jusqu'à ce qu'aucun terminal (ni  $\varepsilon$ ) ne puisse être ajouté à Prem(X).

#### Détermination de Prem(X)

- ① Si  $X \in V_T$  alors  $\text{Prem}(X) = \{X\}$
- ② Si  $X \rightarrow \varepsilon$  est une production alors ajouter  $\varepsilon$  à  $\text{Prem}(X)$
- ③ Si  $X \in V_N$  et  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_k$  est une production avec  $Y_i \in (V_T UV_N)$  alors :
  - Ajouter les éléments de  $\text{Prem}(Y_1)$  sauf  $\varepsilon$  à  $\text{Prem}(X)$
  - S'il existe j ( $j \in \{2, 3, \dots, k\}$ ) tel que : pour tout i ( $i = 1..j-1$ ), on a  $\varepsilon \in \text{Prem}(Y_i)$   
Alors Ajouter les éléments de  $\text{Prem}(Y_j)$  sauf  $\varepsilon$  à  $\text{Prem}(X)$
  - Si pour tout i ( $i = 1..k$ ), on a  $\varepsilon \in \text{Prem}(Y_i)$  Alors Ajouter  $\varepsilon$  à  $\text{Prem}(X)$

#### Exemple 2

Soient les règles de productions suivantes pour une grammaire donnée :

$S \rightarrow ABCe$   
 $A \rightarrow aA \mid \varepsilon$   
 $B \rightarrow bB \mid cB \mid \varepsilon$   
 $C \rightarrow de \mid da \mid dA$

On se propose de déterminer les premiers de S, A B et C en appliquant les règles énoncées précédemment :

$\text{Prem}(C) = \{d\}$   
 $\text{Prem}(B) = \{b, c, \varepsilon\}$   
 $\text{Prem}(A) = \{a, \varepsilon\}$   
 $\text{Prem}(S) = \{a\} \cup \{b, c\} \cup \{d\} = \{a, b, c, d\}$

### 4.2.2 Fonction Suivant (Follow)

Pour chaque non terminal  $A$ ,  $Suivant(A)$  définit l'ensemble des terminaux qui peuvent apparaître immédiatement à la droite de  $A$  dans une chaîne. Ceci signifie que :

$$Suivant(A) = \{x \in V_T / S \xrightarrow{*} \alpha A x \beta, \alpha, \beta \in (V_T \cup V_N)^*\}$$

#### Exemple 1

Soient les règles de productions suivantes pour une grammaire donnée :

$$S \rightarrow aAB \mid aAd$$

$$B \rightarrow bB \mid c$$

On se propose de déterminer les suivants du symbole  $A$ . On constate, d'après la première règle de production, que  $A$  peut être suivi par  $B$  ou  $d$ . D'après la deuxième règle,  $B$  peut commencer par  $b$  ou  $c$ . On peut donc déduire que :  **$Suiv(A) = \{b, c, d\}$**

Pour calculer les suivants pour tous les non terminaux, il faut appliquer les trois règles suivantes jusqu'à ce qu'aucun terminal ne puisse être ajouté aux ensembles  $Suiv$ .

#### Détermination des Suivants

- ① Ajouter  $\$$  à  $Suiv(S)$  où  $S$  est l'axiome.
- ② Pour chaque production  $A \rightarrow \alpha B \beta$ , le contenu de  $Prem(\beta)$  sauf  $\epsilon$  est ajouté à  $Suiv(B)$
- ③ S'il existe une production  $A \rightarrow \alpha B$  ou une production  $A \rightarrow \alpha B \beta$  telle que  $Prem(\beta)$  contient  $\epsilon$  ( $\beta \xrightarrow{*} \epsilon$ )  
Alors les éléments de  $Suiv(A)$  sont ajoutés à  $Suiv(B)$

#### Exemple 2

Soient les règles de productions suivantes pour une grammaire donnée :

$$S \rightarrow aSb \mid cd \mid SAe$$

$$A \rightarrow aAdB \mid \epsilon$$

$$B \rightarrow bb$$

On se propose de déterminer les suivants de  $S$ ,  $A$  et  $B$  en appliquant les règles énoncées précédemment :

$$Suiv(S) = \{\$, b, a, e\}$$

$$Suiv(A) = \{e, d\}$$

$$Suiv(B) = \{e, d\}$$

## 4.3 Construction de la table d'analyse prédictive

L'idée principale de l'algorithme de construction de la table d'analyse prédictive  $M$  est que, dans le cas où on a  $A \rightarrow \alpha$  et  $a \in Prem(\alpha)$  alors l'analyseur développera  $A$  en  $\alpha$  chaque fois que le symbole d'entrée courant est  $a$ .

Un problème se pose quand  $\alpha \rightarrow \epsilon$  ou  $\alpha \xrightarrow{*} \epsilon$ , dans ce cas, il faudra développer  $A$  en  $\alpha$  si le symbole d'entrée courant est dans  $Suiv(A)$  ou si le  $\$$  a été atteint et que  $\$ \in Suiv(A)$ .

### Etapes de l'algorithme de construction de la table d'analyse prédictive M

- ① Pour chaque production  $A \rightarrow \alpha$  de la grammaire, exécuter les étapes ② et ③
- ② Pour chaque terminal  $a \in \text{Prem}(\alpha)$ , ajouter  $A \rightarrow \alpha$  à  $M[A, a]$
- ③ Si  $\varepsilon \in \text{Prem}(\alpha)$  Alors ajouter  $A \rightarrow \alpha$  à  $M[A, b]$  pour chaque  $b \in \text{Suiv}(A)$   
Si  $\varepsilon \in \text{Prem}(\alpha)$  et  $\$ \in \text{Suiv}(A)$  Alors ajouter  $A \rightarrow \alpha$  à  $M[A, \$]$
- ④ Affecter *Erreur* aux entrées qui restent vides dans la table

## 4.4 Programme d'analyse prédictive

Le programme d'analyse prédictive contrôle l'analyse syntaxique en considérant le symbole en sommet de pile X et le caractère courant **a** dans le tampon d'entrée (chaîne à analyser). Ainsi, l'action de l'analyseur est déterminée par X et a, il y a trois cas possibles comme le montre l'algorithme suivant.

### Algorithme d'analyse prédictive

**1<sup>er</sup> cas** : Si  $X=a=\$$  Alors l'analyseur s'arrête : analyse réussie, chaîne acceptée

**2<sup>ème</sup> cas** : Si X est un terminal  
Si  $X=a \neq \$$  Alors dépiler X et avancer dans la chaîne à analyser  
Si  $X \neq a$  Alors Erreur

**3<sup>ème</sup> cas** : Si X est un non terminal Alors utiliser  $M[X, a]$   
Si  $M[X, a]$  contient **une** production  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_N$   
Alors  
Dépiler X  
Empiler  $Y_N$  puis  $Y_{N-1}$  puis ...  $Y_2$  puis  $Y_1$   
Emettre en sortie la production  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_N$   
Si  $M[X, a]$  est vide Alors Erreur

## 4.5 Exemples d'application de l'analyse prédictive

### 4.5.1 Exemple 1

Soient les règles de productions suivantes pour une grammaire donnée dont l'axiome est E :

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \varepsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \varepsilon$   
 $F \rightarrow (E) \mid id$

On se propose de construire la table d'analyse prédictive puis d'analyser les chaînes  $id+id*id$  et  $id++id$  en utilisant la méthode d'analyse prédictive non récursive.

Pour cela, on commence d'abord par la détermination des premiers et des suivants pour chaque symbole non terminal de la grammaire afin de faciliter la construction de la table d'analyse. Par la suite, le programme d'analyse va être appliqué pour analyser chacune des chaînes demandées.

### Détermination des premiers et des suivants

Après l'application des algorithmes présentés dans la section 4.2 (pages 36-37), on obtient :

	Prem	Suiv
E	(, id	\$, )
E'	+, ε	\$, )
T	(, id	+, \$, )
T'	*, ε	+, \$, )
F	(, id	*, +, \$, )

### Construction de la table d'analyse M

Après l'application de l'algorithme présenté dans la section 4.3 (pour chacune des productions  $A \rightarrow \alpha$  de la grammaire), on obtient la table M dont les lignes correspondent aux symboles non terminaux (respecter l'ordre d'apparition dans les parties gauches des règles de production) et les colonnes correspondent aux symboles terminaux en plus du \$ (pour les terminaux, respecter l'ordre d'apparition dans les parties droites des règles de production):

Productions de la grammaire

$E \rightarrow TE'$       prem ( $TE'$ ) = { (, id }  
 $E' \rightarrow +TE'$       prem ( $+TE'$ ) = { + }  
 $E' \rightarrow \varepsilon$       suiv ( $E'$ ) = { \$, ) }  
 $T \rightarrow FT'$       prem ( $FT'$ ) = { (, id }  
 $T' \rightarrow *FT'$       prem ( $*FT'$ ) = { \* }  
 $T' \rightarrow \varepsilon$       suiv ( $T'$ ) = { +, \$, ) }  
 $F \rightarrow (E)$       prem ( $(E)$ ) = { ( }  
 $F \rightarrow id$       prem ( $id$ ) = { id }

	+	*	(	)	id	\$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

### Analyse de la chaîne id+id\*id

Après l'application de l'algorithme d'analyse prédictive présenté dans la section 4.4 (page 38), on obtient les transitions effectuées par un analyseur prédictif sur la chaîne d'entrée id+id\*id, comme suit:

Pile	Entrée	Sortie
\$E	id+id*id\$	<b><math>E \rightarrow TE'</math></b>
\$E' T	id+id*id\$	<b><math>T \rightarrow FT'</math></b>
\$E' T' F	id+id*id\$	<b><math>F \rightarrow id</math></b>
\$E' T' id	id+id*id\$	Comparer
\$E' T'	+id*id\$	<b><math>T' \rightarrow \varepsilon</math></b>
\$E'	+id*id\$	<b><math>E' \rightarrow +TE'</math></b>
\$E' T +	+id*id\$	Comparer
\$E' T	id*id\$	<b><math>T \rightarrow FT'</math></b>
\$E' T' F	id*id\$	<b><math>F \rightarrow id</math></b>
\$E' T' id	id*id\$	Comparer
\$E' T'	*id\$	<b><math>T' \rightarrow *FT'</math></b>
\$E' T' F *	*id\$	Comparer
\$E' T' F	id\$	<b><math>F \rightarrow id</math></b>
\$E' T' id	id\$	Comparer
\$E' T'	\$	<b><math>T' \rightarrow \varepsilon</math></b>
\$E'	\$	<b><math>E' \rightarrow \varepsilon</math></b>
\$	\$	Analyse réussie, chaîne acceptée

L'arbre syntaxique de la chaîne id+id\*id peut être déduit directement à partir de la colonne des sorties (productions en gras) comme le montre la figure 4.2.



On remarque que  $M[A, c]$  est une case contenant deux règles de production, donc il y a deux développements possibles :  $A \rightarrow c$  ou  $A \rightarrow cd$ . Dans ce genre de situations (entrées dans  $M$  définies de façon multiple), on ne peut pas utiliser la méthode d'analyse prédictive et on dit que **la grammaire n'est pas LL(1)**.

## 4.6 Grammaire LL(1)

L'algorithme d'analyse prédictive n'est pas applicable si la table d'analyse contient des entrées multiples (plusieurs productions pour une même case  $M[X, a]$ ) car on ne peut pas savoir quelle production appliquer.

**Définition :** On appelle grammaire LL(1) une grammaire pour laquelle la table d'analyse prédictive ne contient aucune case définie de façon multiple, chaque case de la table contient **au plus** une règle de production.

**L :** Left to right scanning (on parcourt ou on analyse la chaîne en entrée de la gauche vers la droite)

**L :** Leftmost derivation (on utilise les dérivations gauches)

**1 :** on utilise **un seul** symbole d'entrée de prévision à chaque étape nécessitant la prise d'une décision d'action d'analyse.

### 4.6.1 Conditions pour qu'une grammaire soit LL(1)

On peut montrer formellement qu'une grammaire est LL(1), si et seulement si, à chaque fois que l'on a une production de la forme  $A \rightarrow \alpha | \beta$ , les trois conditions suivantes sont vérifiées :

**C1 :** Pour aucun terminal  $a$ ,  $\alpha$  et  $\beta$  ne se dérivent toutes les deux en des chaînes commençant par  $a$ .  
Autrement dit :  $\text{Prem}(\alpha) \cap \text{Prem}(\beta) = \emptyset$ .

**C2 :** Une des chaînes ( $\alpha$  ou  $\beta$ ) au plus peut se dériver en la chaîne vide.  
Autrement dit :  $(\beta \rightarrow^* \epsilon)$  ou bien  $(\alpha \rightarrow^* \epsilon)$  mais pas les deux à la fois.

**C3 :** Si  $(\beta \rightarrow^* \epsilon)$ ,  $\alpha$  ne se dérive pas en une chaîne commençant par un terminal de  $\text{Suiv}(A)$ .  
Autrement dit : **Si**  $(\beta \rightarrow^* \epsilon)$  **alors**  $\text{Prem}(\alpha) \cap \text{Suiv}(A) = \emptyset$ .

#### Exemple 1

On se propose de montrer si la grammaire  $G_1$  ayant les règles de productions suivantes est LL(1) (c'est la grammaire de l'exemple 2, section 4.5.2).

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow cd \mid c \end{aligned}$$

Pour la production  $A \rightarrow cd \mid c$ , on constate que  $\text{Prem}(cd) \cap \text{Prem}(c) = \{c\} \cap \{c\} = \{c\} \neq \emptyset$ , ceci signifie que la condition C1 n'est pas vérifiée donc la grammaire  **$G_1$  n'est pas LL(1)**.

#### Exemple 2

Soient les règles de productions suivantes pour une grammaire  $G_2$  dont l'axiome est  $E$  (c'est la grammaire de l'exemple 1, section 4.5.1):

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

On se propose de montrer si  $G_2$  est une grammaire LL(1).

Pour la production  $E' \rightarrow +TE' \mid \varepsilon$

C1 :  $\text{Prem}(+TE') \cap \text{Prem}(\varepsilon) = \{+\} \cap \{\varepsilon\} = \emptyset$  vérifiée

C2 : vérifiée car  $+TE'$  ne peut pas se dériver en  $\varepsilon$

C3 :  $\text{Prem}(+TE') \cap \text{Suiv}(E') = \{+\} \cap \{\$, \}) = \emptyset$  vérifiée

Pour la production  $T' \rightarrow *FT' \mid \varepsilon$

C1 :  $\text{Prem}(*FT') \cap \text{Prem}(\varepsilon) = \{*\} \cap \{\varepsilon\} = \emptyset$  vérifiée

C2 : vérifiée car  $*FT'$  ne peut pas se dériver en  $\varepsilon$

C3 :  $\text{Prem}(*FT') \cap \text{Suiv}(T') = \{*\} \cap \{+, \$, \}) = \emptyset$  vérifiée

Pour la production  $F \rightarrow (E) \mid \text{id}$

C1 :  $\text{Prem}((E)) \cap \text{Prem}(\text{id}) = \{( \} \cap \{\text{id}\} = \emptyset$  vérifiée

C2 : vérifiée car  $(E) \rightarrow^* \varepsilon$  et  $\text{id} \rightarrow^* \varepsilon$  sont toutes les deux impossibles

C3 : non applicable (à cause de C2) vérifiée

**Conclusion** : Les trois conditions (C1, C2 et C3) sont vérifiées pour toutes les productions de la forme  $A \rightarrow \alpha \mid \beta$ , donc la grammaire **G2 est LL(1)**.

## 4.6.2 Transformations pour qu'une grammaire devienne LL(1)

**Théorème** : Une grammaire **ambiguë**, ou **récursive à gauche** ou **non factorisée à gauche**, **n'est pas** une grammaire **LL(1)**.

Ainsi, si la table d'analyse prédictive d'une grammaire comporte des cases définies de façon multiple, cette grammaire n'est pas LL(1) mais on peut essayer de la transformer en une grammaire LL(1) en effectuant les transformations suivantes :

1. La rendre non ambiguë. Pour cela, il n'existe malheureusement pas de méthode : une grammaire ambiguë est une grammaire mal conçue. Il faut tenter de refaire la conception de la grammaire (tenir compte de la priorité des opérateurs, associer le premier else au dernier if...).
2. Eliminer la récursivité à gauche si cela est nécessaire (voir section 4.6.2.1).
3. Factoriser les règles de production à gauche si cela est nécessaire (voir section 4.6.2.2).

Par la suite, on peut construire la table d'analyse prédictive de la grammaire obtenue après transformations. Si aucune case n'est définie de manière multiple, alors la grammaire obtenue est LL(1) et la méthode d'analyse prédictive peut être appliquée.

Malheureusement, il existe des grammaires pour lesquelles il n'y a pas de transformation les rendant LL(1), dans ce cas, la méthode d'analyse prédictive ne peut être appliquée et il faudra utiliser une autre méthode d'analyse.

### 4.6.2.1 Elimination de la récursivité à gauche

**Définition** : Une grammaire est récursive à gauche si elle contient un non terminal A tel que  $A \xrightarrow{*} A\alpha$  ( $\alpha$  étant une chaîne quelconque).



Les méthodes d'analyse descendante ne peuvent pas fonctionner avec une grammaire récursive à gauche, il faut donc la transformer en une grammaire équivalente en éliminant la récursivité à gauche.

### Elimination de la récursivité à gauche immédiate

Dans le **cas le plus simple**, il s'agit d'une récursivité à gauche immédiate avec la présence d'un non terminal A et une production de la forme  $A \rightarrow A\alpha$ . Pour éliminer la récursivité à gauche immédiate, il faut remplacer toute production de la forme :

$$A \rightarrow A\alpha \mid \beta \quad \text{par :} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}$$

D'une **manière plus générale**, pour éliminer la récursivité à gauche immédiate, il faut procéder comme suit :

1. Regrouper les A-productions de la façon suivante :  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_M \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_N$
2. Remplacer les A-productions par : 
$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_N A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_M A' \mid \varepsilon$$

### Exemple 1

Soit la grammaire  $G_1$  ayant les règles de production suivantes :

$$\begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

Après l'élimination de la récursivité à gauche, on obtient une grammaire  $G'_1$  équivalente à  $G_1$ , dont les règles de production sont :

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \varepsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \varepsilon \\ F \rightarrow (E) \mid \text{id} \end{array}$$

### Exemple 2

Soit la grammaire  $G_2$  ayant les règles de production suivantes :

$$\begin{array}{l} S \rightarrow ScA \mid B \\ A \rightarrow Aa \mid \varepsilon \\ B \rightarrow Bb \mid d \mid e \end{array}$$

Après l'élimination de la récursivité à gauche, on obtient une grammaire  $G'_2$  équivalente à  $G_2$ , dont les règles de production sont :

$$\begin{array}{l} S \rightarrow BS' \\ S' \rightarrow cAS' \mid \varepsilon \\ A \rightarrow A' \\ A' \rightarrow aA' \mid \varepsilon \\ B \rightarrow dB' \mid eB' \\ B' \rightarrow bB' \mid \varepsilon \end{array}$$

Il est aussi possible que la récursivité gauche ne soit pas immédiate, mais qu'on ne puisse la détecter qu'après avoir effectué des dérivations. D'une manière encore plus générale, pour éliminer la récursivité à gauche, on peut appliquer l'algorithme suivant à condition que la grammaire soit propre, autrement dit, ne contienne pas de production  $A \rightarrow \varepsilon$  :

**Algorithme** Elimination de récursivité gauche d'une grammaire propre;

**Début**

Ordonner les non terminaux  $A_1, A_2, \dots, A_N$  ;

**Pour**  $i=1$  à  $N$  **Faire**

**Début**

**Pour**  $k=1$  à  $i-1$  **Faire**

**Début**

Remplacer chaque production de la forme

$A_i \rightarrow A_k \alpha \mid \lambda$  où  $A_k \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_p$

par  $A_i \rightarrow \beta_1 \alpha \mid \beta_2 \alpha \mid \dots \mid \beta_p \alpha \mid \lambda$

**Fin;**

Éliminer les récursivités à gauche immédiate des productions  $A_i$

**Fin ;**

**Fin ;**

### Exemple 3

Soit la grammaire  $G_3$  ayant les règles de production suivantes :

$Z \rightarrow Aa \mid z$

$A \rightarrow Ac \mid Zd$

On commence par ordonner  $Z$  et  $A$  ( $A_1=Z, A_2=A$ ).

Pour  $i=1$ , pas de récursivité immédiate dans  $Z \rightarrow Aa \mid z$

Pour  $i=2$  et  $k=1$ , on obtient  $A \rightarrow Ac \mid Aad \mid zd$

on élimine la récursivité immédiate :

$A \rightarrow zdA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

On obtient ainsi une grammaire  $G'_3$  équivalente à  $G_3$ , non récursive à gauche et dont les règles de production sont :

$Z \rightarrow Aa \mid z$

$A \rightarrow zdA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

### Exemple 4

Soit la grammaire  $G_4$  ayant les règles de production suivantes :

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid BA \mid c$

$B \rightarrow SSc \mid a$

On commence par ordonner  $S, A, B$  ( $A_1=S, A_2=A, A_3=B$ ).

Pour  $i=1$ , pas de récursivité immédiate dans  $S \rightarrow Aa \mid b$

Pour  $i=2$  et  $k=1$ , on obtient  $A \rightarrow Ac \mid Aad \mid bd \mid BA \mid c$

on élimine la récursivité immédiate

$A \rightarrow bdA' \mid BAA' \mid cA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Pour  $i=3$  et  $k=1$ , on obtient :  $B \rightarrow AaSc \mid bSc \mid a$   
 et  $k=2$  donne :  $B \rightarrow bdA'aSc \mid BAA'aSc \mid cA'aSc \mid bSc \mid a$   
 on élimine la récursivité immédiate  
 $B \rightarrow bdA'aScB' \mid cA'aScB' \mid bScB' \mid aB'$   
 $B' \rightarrow AA'aScB' \mid \varepsilon$

On obtient ainsi une grammaire  $G'_4$  équivalente à  $G_4$ , non récursive à gauche et dont les règles de production sont :

$S \rightarrow Aa \mid b$   
 $A \rightarrow bdA' \mid BAA' \mid cA'$   
 $A' \rightarrow cA' \mid adA' \mid \varepsilon$   
 $B \rightarrow bdA'aScB' \mid cA'aScB' \mid bScB' \mid aB'$   
 $B' \rightarrow AA'aScB' \mid \varepsilon$

### Rappel (grammaire propre)

Une grammaire est dite propre si elle ne contient aucune production  $A \rightarrow \varepsilon$ . Pour rendre une grammaire propre, il faut rajouter une production dans laquelle le  $A$  est remplacé par  $\varepsilon$ , ceci pour chaque  $A$  apparaissant en partie droite d'une production, et pour chaque  $A$  d'une production  $A \rightarrow \varepsilon$ .

### Exemple (grammaire propre)

Soit la grammaire  $G_5$  ayant les règles de production suivantes, on veut rendre cette grammaire propre :

$S \rightarrow aTb \mid aU$   
 $T \rightarrow bTaTA \mid \varepsilon$   
 $U \rightarrow aU \mid b$

On obtient ainsi une grammaire propre  $G'_5$  équivalente à  $G_5$ , dont les règles de production sont :

$S \rightarrow aTb \mid ab \mid aU$   
 $T \rightarrow bTaTA \mid baTA \mid bTaA \mid baA$   
 $U \rightarrow aU \mid b$

## 4.6.2.2 Factorisation à gauche

Cette transformation est un moyen pour obtenir une grammaire convenant à l'analyse prédictive. L'idée de la factorisation à gauche est de réécrire les A-productions du type  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_N \mid \lambda$  de sorte à reporter la décision (quelle production choisir ?) jusqu'à ce que suffisamment de texte soit lu pour faire le bon choix.

Pour effectuer la factorisation à gauche, il faut remplacer :

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_N \mid \lambda$  par  $A \rightarrow \alpha A' \mid \lambda$   
 $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_N$

où  $\alpha \in (V_T \cup V_N)^+$  et  $\lambda$  représente toutes les alternatives qui ne commencent pas par  $\alpha$ .

### Remarque :

Pour chaque non terminal  $A$ , il faut essayer de trouver le plus long préfixe  $\alpha$  commun à deux ou plusieurs des parties droites des A-productions. Il faut refaire l'opération de factorisation jusqu'à ne plus trouver de préfixes communs.

### Exemple 1

Soit la grammaire  $G_1$  ayant les règles de production suivantes :

$$S \rightarrow aC \mid abD \mid aBc$$

Après factorisation à gauche, on obtient une grammaire  $G'_1$  équivalente à  $G_1$ , dont les règles de production sont :

$$S \rightarrow aS'$$

$$S' \rightarrow C \mid bD \mid Bc$$

### Exemple 2

Soit la grammaire  $G_2$  ayant les règles de production suivantes :

$$S \rightarrow aEbS \mid aEbSeB \mid a$$

$$E \rightarrow bcB \mid bca$$

$$B \rightarrow ba$$

Après une première factorisation à gauche, on obtient :

$$S \rightarrow aEbSS' \mid a$$

$$S' \rightarrow \varepsilon \mid eB$$

$$E \rightarrow bcE'$$

$$E' \rightarrow B \mid a$$

$$B \rightarrow ba$$

Finalement, la grammaire  $G'_2$  factorisée à gauche et équivalente à  $G_2$ , aura pour règles de production :

$$S \rightarrow aS''$$

$$S'' \rightarrow EbSS' \mid \varepsilon$$

$$S' \rightarrow \varepsilon \mid eB$$

$$E \rightarrow bcE'$$

$$E' \rightarrow B \mid a$$

$$B \rightarrow ba$$

## SERIE DE TD N°:3 COMPILATION, ANALYSE SYNTAXIQUE DESCENDANTE

### Exercice 1

Déterminez, en donnant des explications, les ensembles de **Premiers** et de **Suivants** de chacun des non terminaux des grammaires suivantes, S étant toujours l'axiome :

$S \rightarrow a A B b$	$S \rightarrow a x B A \mid C$	$S \rightarrow A$	$S \rightarrow AB$
$A \rightarrow A e \mid f$	$A \rightarrow B S \mid y$	$A \rightarrow Ad \mid Ae \mid aB \mid aC$	$A \rightarrow CD \mid CB \mid b$
$B \rightarrow C D$	$B \rightarrow A S C \mid A A \mid z$	$B \rightarrow bBC \mid f$	$B \rightarrow aE \mid aB \mid \varepsilon$
$C \rightarrow g \mid \varepsilon$	$C \rightarrow S S \mid t$	$C \rightarrow g$	$C \rightarrow b$
$D \rightarrow h \mid \varepsilon$			$D \rightarrow De \mid f$
			$E \rightarrow a \mid \varepsilon$

### Exercice 2

Soit la grammaire  $G_1 = (\{a, (, -, )\}, \{A, B, C, D\}, A, P)$

P:	$A \rightarrow -A \mid (A) \mid CB$	1. Déterminer les premiers et les suivants de chaque non terminal.
	$B \rightarrow -A \mid \varepsilon$	2. Etablir la table d'analyse prédictive de $G_1$ .
	$C \rightarrow a D$	3. Analyser, en utilisant la méthode d'analyse prédictive, la chaîne <b>-a(-a)-a</b>
	$D \rightarrow (A) \mid \varepsilon$	et déduire son arbre de dérivation. Analyser aussi la chaîne <b>-a-</b> .

### Exercice 3

Montrer si les grammaires  $G_1, G_2, G_3, G_4$  et  $G_5$  ayant pour axiomes respectifs  $S_1, S_2, S_3, S_4$  et  $S_5$  sont des grammaires LL(1). Pour celles qui ne le sont pas, effectuer les transformations nécessaires pour qu'elles le deviennent et montrez si les nouvelles grammaires obtenues sont LL(1) ou pas.

$G_1: S_1 \rightarrow aAc$	$G_2: S_2 \rightarrow BcBd \mid CdCc$	$G_3: S_3 \rightarrow De \mid hDf \mid Ef \mid hEe$	$G_4: S_4 \rightarrow FG$	$G_5: S_5 \rightarrow aTb \mid \varepsilon$
$A \rightarrow Abb \mid b$	$B \rightarrow \varepsilon$	$D \rightarrow g$	$F \rightarrow i \mid \varepsilon$	$T \rightarrow c S_5 a \mid \varepsilon$
	$C \rightarrow \varepsilon$	$E \rightarrow g$	$G \rightarrow j \mid \varepsilon$	

### Exercice 4

Soit la grammaire  $G_2 = (\{a, b, (, ) , , \}, \{S, T\}, S, P)$ , ayant les règles de production suivantes :

$$\begin{aligned} S &\rightarrow a \mid b \mid (T) \\ T &\rightarrow T,S \mid S \end{aligned}$$

- Montrer si la grammaire  $G_2$  est une grammaire LL(1). Si elle ne l'est pas, effectuer les transformations nécessaires pour qu'elle le devienne. Soit  $G'_2$  la grammaire obtenue après ces éventuelles transformations.
- Déterminer les premiers et les suivants de chaque non terminal de  $G'_2$ .
- Etablir la table d'analyse prédictive de  $G'_2$ .
- Analyser les chaînes **(a,b)** puis **(a,(b))** et **(a,(a,b))** en utilisant l'analyse prédictive et déduire leurs arbres syntaxiques.

### Exercice 5

Soit la grammaire suivante (des expressions booléennes) dont l'axiome est A:

$$\begin{aligned} A &\rightarrow A \text{ ou } B \mid B \\ B &\rightarrow B \text{ et } C \mid C \\ C &\rightarrow \text{non } C \mid D \\ D &\rightarrow (A) \mid \text{vrai} \mid \text{faux} \end{aligned}$$

- Eliminer la récursivité à gauche et factoriser si cela est nécessaire.
- Donner la table d'analyse de la nouvelle grammaire. Est-elle LL(1) ?
- Expliciter le comportement de l'analyseur sur la chaîne **non (vrai ou faux) et vrai**.

### Exercice 6

Soit la grammaire suivante dont l'axiome est S:

$$\begin{aligned} S &\rightarrow Ab \mid a \mid AA \\ A &\rightarrow Sa \mid Ac \mid B \\ B &\rightarrow Sd \end{aligned}$$

- Eliminer la récursivité à gauche de cette grammaire.
- Factoriser les règles obtenues si cela est nécessaire.
- La grammaire obtenue est-elle une grammaire LL(1) ?

## CHAPITRE 5 : ANALYSE SYNTAXIQUE : METHODES ASCENDANTES

### 5.1 Introduction à l'analyse ascendante

Les méthodes ascendantes construisent l'arbre syntaxique de bas en haut, en partant de la chaîne analysée (feuilles de l'arbre), puis, en assemblant, par des réductions, les sous-arbres sous les nouveaux nœuds non terminaux jusqu'à l'axiome (racine de l'arbre).

Le modèle général utilisé en analyse ascendante est le modèle par **décalage-réduction (shift-reduce)** qui autorise deux opérations :

- décaler (shift) : décaler, d'un symbole, le pointeur sur la chaîne d'entrée.
- réduire (reduce) : réduire une chaîne par un non terminal en utilisant une des règles de production, sachant que la chaîne réduite est une suite de terminaux et non terminaux à gauche du pointeur sur l'entrée et finissant sur ce pointeur.

#### Exemple

Soit la grammaire G ayant les règles de production suivantes :

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

On se propose d'analyser la chaîne *abbcde* de manière ascendante.

a b b c d e	décaler
a <b>b</b> b c d e	réduire
a <b>A</b> b c d e	décaler
a A <b>b</b> c d e	décaler
a A b <b>c</b> d e	réduire
a <b>A</b> d e	décaler
a A <b>d</b> e	réduire
a A <b>B</b> e	décaler
a A B <b>e</b>	réduire
<b>S</b>	Analyse réussie

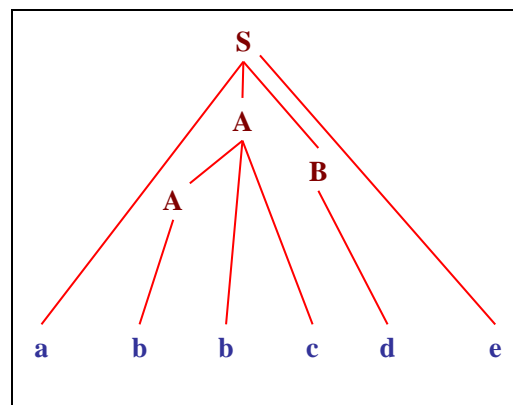


Figure 5.1. Arbre syntaxique obtenu de manière ascendante pour la chaîne *abbcde*

En résumé, on a donc les réductions suivantes, à partir desquelles on peut déduire, de manière ascendante, l'arbre syntaxique correspondant à la chaîne *abbcde* (voir figure 5.1) :

a b b c d e  
a A b c d e  
a A d e  
a A B e  
S

On constate que la séquence de réductions précédentes correspond, en sens inverse, aux dérivations droites suivantes :  $S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abbcde$

**Remarque :** Il serait intéressant de disposer d'un moyen automatique pour choisir s'il faut effectuer un décalage ou une réduction et quelle réduction choisir lorsque le pointeur d'entrée se trouve sur une position donnée.

## 5.2 Introduction aux analyseurs LR

L'analyse LR (ou LR(k)) est une technique générale efficace d'analyse syntaxique ascendante, basée sur le modèle par décalage-réduction et qui peut être utilisée pour analyser une large classe de grammaires non contextuelles.

**L :** Left to right scanning (on parcourt ou on analyse la chaîne en entrée de la gauche vers la droite)

**R :** constructing a **R**ightmost derivation in reverse (en construisant une dérivation droite inverse)

**k :** on utilise **k** symboles d'entrée de prévision à chaque étape nécessitant la prise d'une décision d'action d'analyse. Quand k est omis, il est supposé égal à 1.

Les analyseurs LR comportent :

- Un algorithme d'analyse commun aux différentes méthodes LR.
- Une table d'analyse dont le contenu diffère selon le type d'analyseur LR.

On distingue trois techniques de construction de tables d'analyse LR pour une grammaire donnée :

- **Simple LR (SLR) :** qui est la plus simple à implémenter, mais la moins puissante des trois.
- **LR canonique :** qui est la plus puissante, mais aussi la plus coûteuse.
- **LookAhead LR (LALR) :** qui a une puissance et un coût intermédiaires entre les deux autres et peut être appliquée à la majorité des grammaires de langages de programmation.

### 5.2.1 Modèle d'analyseur LR

Un analyseur LR peut être assimilé à un automate à pile, il est modélisé, comme le montre la figure 5.2, par un tampon d'entrée, un flot de sortie, une pile, un programme d'analyse (le même pour tous les analyseurs LR) et des tables d'analyse divisées en deux parties : Action et Successeur (Goto).

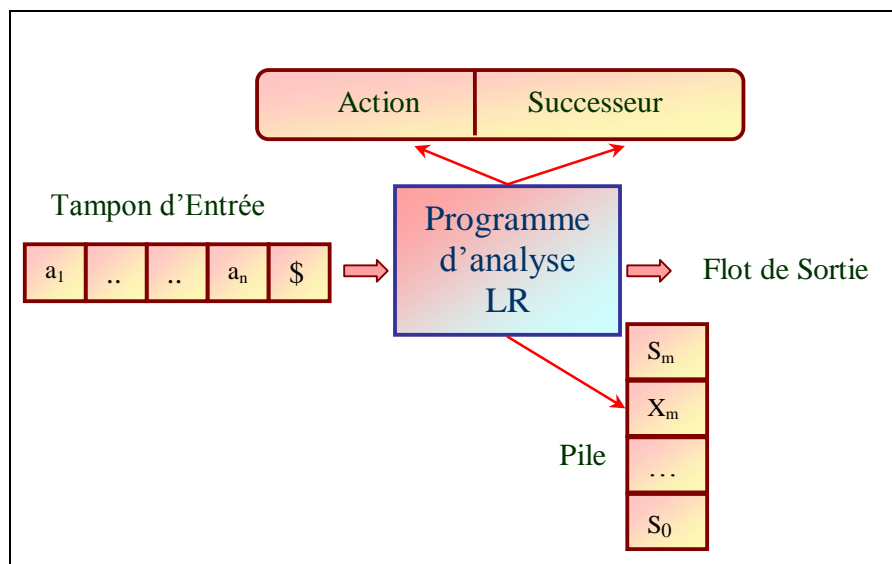


Figure 5.2. Modèle d'un analyseur LR

Le programme d'analyse range dans la pile des chaînes de la forme  $S_0X_1S_1X_2S_2...X_mS_m$ . Chaque  $X_i$  est un symbole de la grammaire ( $X_i \in (V_T \cup V_N)$ ) et chaque  $S_i$  est un état (state) de l'analyseur qui résume l'information contenue dans la pile au dessous de lui, la combinaison du numéro de l'état en sommet de pile et du symbole d'entrée courant est utilisé pour indiquer les tables et déterminer l'action à effectuer (décaler ou réduire).

Les tables d'analyse contiennent deux parties : une fonction d'action d'analyse (action) et une fonction de transfert (successeur). L'information contenue dans ces champs représente la différence entre les analyseurs LR.

### 5.2.2 Algorithme d'analyse LR

Le programme d'analyse LR détermine  $S_m$  (l'état en sommet de pile) et  $a_i$  (le symbole terminal d'entrée courant). Ceci correspond à la configuration de l'analyseur  $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_{i+1} \dots a_n \$)$  sachant que la configuration initiale est  $(S_0, a_1 a_2 \dots a_n \$)$ . L'analyseur consulte **Action**  $[S_m, a_i]$ , dans la table des actions, qui peut avoir une des quatre valeurs : *décaler*, *réduire*, *accepter* ou *erreur*.

#### 1<sup>er</sup> cas : Action $[S_m, a_i]$ = décaler S (shift S) :

L'analyseur empile  $a_i$  et  $S$ ,  $a_{i+1}$  devient le symbole d'entrée courant. La configuration de l'analyseur devient donc :  $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$ .

#### 2<sup>ème</sup> cas : Action $[S_m, a_i]$ = réduire par $A \rightarrow \beta$ :

L'analyseur dépile  $2r$  symboles,  $r$  symboles d'états et  $r$  symboles de grammaire, avec  $r = |\beta|$  = longueur de  $\beta$ .  $S_{m-r}$  devient le nouveau sommet de pile, ensuite l'analyseur empile  $A$  (partie gauche de la production  $A \rightarrow \beta$ ) et  $S$  (entrée pour Successeur $[S_{m-r}, A]$ ).

#### Remarques :

Le symbole d'entrée courant ne change pas avec une réduction.

La séquence dépilée  $X_{m-r+1} \dots X_m$  correspond à  $\beta$ . La configuration de l'analyseur devient donc :

$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_{i+1} \dots a_n \$)$ .

#### 3<sup>ème</sup> cas : Action $[S_m, a_i]$ = accepter :

L'analyse est terminée en acceptant la chaîne analysée.

#### 4<sup>ème</sup> cas : Action $[S_m, a_i]$ = erreur :

L'analyse est terminée en rejetant la chaîne analysée.

### Résumé de l'algorithme d'analyse LR

**Entrée** : Chaîne  $w$  et table d'analyse LR

**Sortie** : Résultat de l'analyse ascendante de  $w$  si  $w \in L(G)$  (arbre syntaxique) ou échec si  $w \notin L(G)$ .

**Méthode** : on commence avec  $S_0$  dans la pile,  $w\$$  dans le tampon d'entrée, le pointeur source  $ps$  est initialisé au début de  $w$ . L'analyseur exécute la boucle *Répéter* jusqu'à ce qu'il rencontre une action *Accepter* ou *Erreur*.

<p><b>Algorithme</b> Analyse LR;  <b>Répéter</b> indéfiniment  <b>Début</b>          Soit <math>S</math> l'état en sommet de pile et <math>a</math> le symbole pointé par <math>ps</math>  <b>Si</b> Action <math>[S, a]</math> = Décaler <math>S'</math> <b>Alors</b>            <b>Début</b>              Empiler <math>a</math> puis <math>S'</math> ;              Avancer <math>ps</math> ;            <b>Fin</b> ;  <b>Sinon Si</b> Action <math>[S, a]</math> = Réduire par <math>A \rightarrow \beta</math> <b>Alors</b>            <b>Début</b>              Dépiler <math>2 \beta </math> symboles ;              Soit <math>S'</math> le nouveau sommet de pile ;              Empiler <math>A</math> puis Successeur<math>[S', A]</math> ;              Emettre en sortie <math>A \rightarrow \beta</math> ;            <b>Fin</b>            <b>Sinon Si</b> Action <math>[S, a]</math> = Accepter <b>Alors</b> Return <b>Sinon</b> Erreur() ;  <b>Fin</b> ;</p>
---



### 5.2.3 Exemple d'application de l'algorithme d'analyse LR

Soit la grammaire G ayant les règles de production suivantes :

$E \rightarrow E+T$	①	$T \rightarrow F$	④
$E \rightarrow T$	②	$F \rightarrow (E)$	⑤
$T \rightarrow T*F$	③	$F \rightarrow id$	⑥

Soit la table d'analyse LR supposée déjà construite :

Etat	Action						Successeur		
	+	*	(	)	id	\$	E	T	F
0			d <sub>4</sub>		d <sub>5</sub>		1	2	3
1	d <sub>6</sub>					acc			
2	r <sub>2</sub>	d <sub>7</sub>		r <sub>2</sub>		r <sub>2</sub>			
3	r <sub>4</sub>	r <sub>4</sub>		r <sub>4</sub>		r <sub>4</sub>			
4			d <sub>4</sub>		d <sub>5</sub>		8	2	3
5	r <sub>6</sub>	r <sub>6</sub>		r <sub>6</sub>		r <sub>6</sub>			
6			d <sub>4</sub>		d <sub>5</sub>			9	3
7			d <sub>4</sub>		d <sub>5</sub>				10
8	d <sub>6</sub>			d <sub>11</sub>					
9	r <sub>1</sub>	d <sub>7</sub>		r <sub>1</sub>		r <sub>1</sub>			
10	r <sub>3</sub>	r <sub>3</sub>		r <sub>3</sub>		r <sub>3</sub>			
11	r <sub>5</sub>	r <sub>5</sub>		r <sub>5</sub>		r <sub>5</sub>			

L'action d<sub>i</sub> signifie décaler et empiler l'état i.

L'action r<sub>j</sub> signifie réduire par la production dont le numéro est j.

L'action acc signifie accepter la chaîne analysée.

Une entrée vide correspond à une erreur.

On se propose maintenant d'analyser la chaîne id \*id+id.

Pile	Entrée	Sortie
S <sub>0</sub>	id*id+id\$	Décaler (d <sub>5</sub> )
S <sub>0</sub> idS <sub>5</sub>	*id+id\$	Réduire F→id
S <sub>0</sub> FS <sub>3</sub>	*id+id\$	Réduire T→F
S <sub>0</sub> TS <sub>2</sub>	*id+id\$	Décaler (d <sub>7</sub> )
S <sub>0</sub> TS <sub>2</sub> *S <sub>7</sub>	id+id\$	Décaler(d <sub>5</sub> )
S <sub>0</sub> TS <sub>2</sub> *S <sub>7</sub> idS <sub>5</sub>	+id\$	Réduire F→id
S <sub>0</sub> TS <sub>2</sub> *S <sub>7</sub> FS <sub>10</sub>	+id\$	Réduire T→T*F
S <sub>0</sub> TS <sub>2</sub>	+id\$	Réduire E→T
S <sub>0</sub> ES <sub>1</sub>	+id\$	Décaler (d <sub>6</sub> )
S <sub>0</sub> ES <sub>1</sub> +S <sub>6</sub>	id\$	Décaler (d <sub>5</sub> )
S <sub>0</sub> ES <sub>1</sub> +S <sub>6</sub> idS <sub>5</sub>	\$	Réduire F→id
S <sub>0</sub> ES <sub>1</sub> +S <sub>6</sub> FS <sub>3</sub>	\$	Réduire T→F
S <sub>0</sub> ES <sub>1</sub> +S <sub>6</sub> TS <sub>9</sub>	\$	Réduire E→E+T
S <sub>0</sub> ES <sub>1</sub>	\$	Accepter

## 5.3 Construction de table d'analyse SLR

La construction d'une table d'analyse SLR (LR(0) ou SLR(1)) à partir d'une grammaire est la méthode la plus simple à implémenter (parmi les 3 méthodes SLR, LALR et LR canonique) mais c'est la moins puissante du point de vue du nombre de grammaire pour lesquelles elle réussit. Elle constitue un bon point de départ pour l'étude de l'analyse LR. Le principe de la méthode SLR est de construire, à partir de la grammaire un automate fini déterministe qui reconnaît les préfixes viables de G (préfixes pouvant apparaître sur la pile) puis de transformer cet automate en une table d'analyse.

La construction des analyseurs SLR, pour une grammaire donnée, est basée sur la collection d'ensembles d'items LR(0). Pour construire cette collection, on définit une **grammaire augmentée**, une fonction **Fermeture** et une fonction **Transition**. Dans la section 5.3.1, on commence par définir les concepts nécessaires à la construction de collection d'ensembles d'items LR(0) et de table d'analyse SLR.

### 5.3.1 Concepts de base

#### 5.3.1.1 Items LR(0)

Un item LR(0) d'une grammaire G est une production de G avec un point (.) repérant une position de sa partie droite.

##### Exemples

La production  $A \rightarrow XYZ$  fournit 4 items :  $A \rightarrow .XYZ$ ,  $A \rightarrow X.YZ$ ,  $A \rightarrow XY.Z$  et  $A \rightarrow XYZ.$

La production  $A \rightarrow \varepsilon$  fournit un seul item  $A \rightarrow .$

##### Remarque

Intuitivement, un item indique la «quantité» de partie droite qui a été reconnue à un instant donné de l'analyse. Par exemple, l'item  $A \rightarrow X.YZ$  indique qu'on vient de voir en entrée une chaîne dérivée de X et qu'on espère maintenant voir une chaîne dérivée de YZ.

#### 5.3.1.2 Grammaire augmentée

Si G est une grammaire d'axiome S, alors la grammaire augmentée de G (notée G') aura un nouvel axiome S' et une nouvelle règle de production  $S' \rightarrow S$ . Le but de l'ajout de cette production est d'indiquer à l'analyseur quand il doit s'arrêter et annoncer l'acceptation de la chaîne d'entrée (quand l'analyseur est sur le point de réduire S par S').

#### 5.3.1.3 Fonction Fermeture d'un ensemble d'items LR(0)

Soit I un ensemble d'items pour une grammaire G. Fermeture (I) est l'ensemble d'items construit à partir de I par l'application des 2 règles suivantes :

**Règle 1** : initialement placer chaque item de I dans  $\text{Ferm}(I)$

**Règle 2** : si  $A \rightarrow \alpha.B\beta \in \text{Ferm}(I)$  et  $B \rightarrow \delta$  est une production de G, alors ajouter  $B \rightarrow .\delta$  à  $\text{Ferm}(I)$ , s'il n'y existe pas déjà. La règle 2 doit être appliquée jusqu'à ne plus pouvoir ajouter de nouveaux items à  $\text{Ferm}(I)$ .

Ainsi, la fonction  $\text{Ferm}(I)$  peut être définie de la manière suivante :

**Fonction**  $\text{Ferm}(I)$ ;  
**Début**  
 $J := I$  ;  
**Répéter**  
**Pour** chaque item  $A \rightarrow \alpha.B\beta \in J$  et chaque production  $B \rightarrow \delta$  telle que  $B \rightarrow .\delta \notin J$   
**Faire** Ajouter  $B \rightarrow .\delta$  à J  
**Jusqu'à** ce qu'aucun item ne puisse être ajouté à J ;  
 $\text{Ferm}(I) := J$ ;  
**Fin** ;

##### Exemple

Soit la grammaire G, ayant les règles de production suivantes :

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

Soit l'ensemble d'items  $I = \{[E' \rightarrow .E]\}$ . On se propose de calculer  $\text{Ferm}(I)$ . En appliquant les deux règles précédentes, on trouve :  $\text{Ferm}(I) = \{ [E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .\text{id}] \}$

### 5.3.1.4 Fonction Transition

Soit  $I$  un ensemble d'items et  $X$  un symbole de la grammaire ( $X \in V_T \cup V_N$ ). Transition  $(I, X)$  est définie comme étant la fermeture de tous les items  $[A \rightarrow \alpha X \beta]$  tels que  $[A \rightarrow \alpha \cdot X \beta] \in I$ .

#### Exemple

En utilisant la grammaire précédente, on se propose de déterminer, pour  $I = \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ , la fonction  $\text{Trans}(I, +)$ .

$\text{Trans}(I, +) = \text{Ferm}([E \rightarrow E \cdot + T]) = \{[E \rightarrow E \cdot + T], [T \rightarrow \cdot T * F], [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}$

### 5.3.2 Construction de la collection d'ensembles d'items LR(0)

La collection canonique d'ensembles d'items LR(0) correspond aux états de l'AFD permettant de reconnaître les préfixes viables de la grammaire. La détermination de cette collection constitue la base de la construction des tables d'analyses SLR. L'algorithme suivant permet de déterminer  $C$  : la collection canonique d'ensemble d'items LR(0) pour une grammaire augmentée  $G'$

**Procédure** CollectionEnsembleItems;

**Début**

$C := \{\text{Ferm}(\{[S' \rightarrow \cdot S]\})\}$  ;

**Répéter**

**Pour** chaque ensemble d'items  $I$  de  $C$  et pour chaque symbole de la grammaire  $X$  tel que  $\text{Trans}(I, X)$  soit non vide et non encore dans  $C$

**Faire** ajouter  $\text{Trans}(I, X)$  à  $C$

**Jusqu'à** ce qu'aucun ensemble d'items ne puisse être ajouté à  $C$ ;

**Fin** ;

#### Exemple

On se propose de déterminer la collection canonique d'ensembles d'items de la grammaire augmentée suivante :

$E' \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \text{id}$

$I_0 = \text{Ferm} \{[E' \rightarrow \cdot E]\}$   
 $= E' \rightarrow \cdot E$   
 $E \rightarrow \cdot E + T$   
 $E \rightarrow \cdot T$   
 $T \rightarrow \cdot T * F$   
 $T \rightarrow \cdot F$   
 $F \rightarrow \cdot (E)$   
 $F \rightarrow \cdot \text{id}$

$I_1 = \text{Trans}(I_0, E) = \text{Ferm} \{[E' \rightarrow E \cdot], [E' \rightarrow E \cdot + T]\}$   
 $= E' \rightarrow E \cdot$   
 $E \rightarrow E \cdot + T$

$I_2 = \text{Trans}(I_0, T) = \text{Ferm} \{[E \rightarrow T \cdot], [T \rightarrow T \cdot * F]\}$   
 $= E \rightarrow T \cdot$   
 $T \rightarrow T \cdot * F$

$I_3 = \text{Trans}(I_0, F) = \text{Ferm} \{[T \rightarrow F \cdot]\}$   
 $= T \rightarrow F \cdot$

$$\begin{aligned}
I_4 &= \text{Trans}(I_0, () = \text{Ferm} \{[F \rightarrow (.E)]\} \\
&= F \rightarrow (.E) \\
&\quad E \rightarrow .E + T \\
&\quad E \rightarrow .T \\
&\quad T \rightarrow .T * F \\
&\quad T \rightarrow .F \\
&\quad F \rightarrow .(E) \\
&\quad F \rightarrow .id
\end{aligned}$$

$$\begin{aligned}
I_5 &= \text{Trans}(I_0, id) = \text{Ferm} \{[F \rightarrow id.]\} \\
&= F \rightarrow id.
\end{aligned}$$

$$\begin{aligned}
I_6 &= \text{Trans}(I_1, +) = \text{Ferm} \{[E \rightarrow E + .T]\} \\
&= E \rightarrow E + .T \\
&\quad T \rightarrow .T * F \\
&\quad T \rightarrow .F \\
&\quad F \rightarrow .(E) \\
&\quad F \rightarrow .id
\end{aligned}$$

$$\begin{aligned}
I_7 &= \text{Trans}(I_2, *) = \text{Ferm} \{[T \rightarrow T * .F]\} \\
&= T \rightarrow T * .F \\
&\quad F \rightarrow .(E) \\
&\quad F \rightarrow .id
\end{aligned}$$

$$\begin{aligned}
I_8 &= \text{Trans}(I_4, E) = \text{Ferm} \{[F \rightarrow (E.)], [E \rightarrow E + T]\} \\
&= F \rightarrow (E.) \\
&\quad E \rightarrow E + T
\end{aligned}$$

$$I_2 = \text{Trans}(I_4, T) = \text{Ferm} \{[E \rightarrow T.], [T \rightarrow T * F]\}$$

$$I_3 = \text{Trans}(I_4, F) = \text{Ferm} \{[T \rightarrow F.]\}$$

$$I_4 = \text{Trans}(I_4, () = \text{Ferm} \{[F \rightarrow (.E)]\}$$

$$I_5 = \text{Trans}(I_4, id) = \text{Ferm} \{[F \rightarrow id.]\}$$

$$\begin{aligned}
I_9 &= \text{Trans}(I_6, T) = \text{Ferm} \{[E \rightarrow E + T.], [T \rightarrow T * F]\} \\
&= E \rightarrow E + T. \\
&\quad T \rightarrow T * F
\end{aligned}$$

$$I_3 = \text{Trans}(I_6, F) = \text{Ferm} \{[T \rightarrow F.]\}$$

$$I_4 = \text{Trans}(I_6, () = \text{Ferm} \{[F \rightarrow (.E)]\}$$

$$I_5 = \text{Trans}(I_6, id) = \text{Ferm} \{[F \rightarrow id.]\}$$

$$\begin{aligned}
I_{10} &= \text{Trans}(I_7, F) = \text{Ferm} \{[T \rightarrow T * F.]\} \\
&= T \rightarrow T * F.
\end{aligned}$$

$$I_4 = \text{Trans}(I_7, () = \text{Ferm} \{[F \rightarrow (.E)]\}$$

$$I_5 = \text{Trans}(I_7, id) = \text{Ferm} \{[F \rightarrow id.]\}$$

$$\begin{aligned}
I_{11} &= \text{Trans}(I_8, ) = \text{Ferm} \{[F \rightarrow (E.)]\} \\
&= F \rightarrow (E.).
\end{aligned}$$

$$I_6 = \text{Trans}(I_8, +) = \text{Ferm} \{[E \rightarrow E + .T]\}$$

$$I_7 = \text{Trans}(I_9, *) = \text{Ferm} \{[T \rightarrow T * .F]\}$$

La fonction de transition pour la collection canonique d'ensembles d'items de la grammaire augmentée considérée dans cet exemple est donnée sous forme d'un AFD représenté par la figure 5.3.

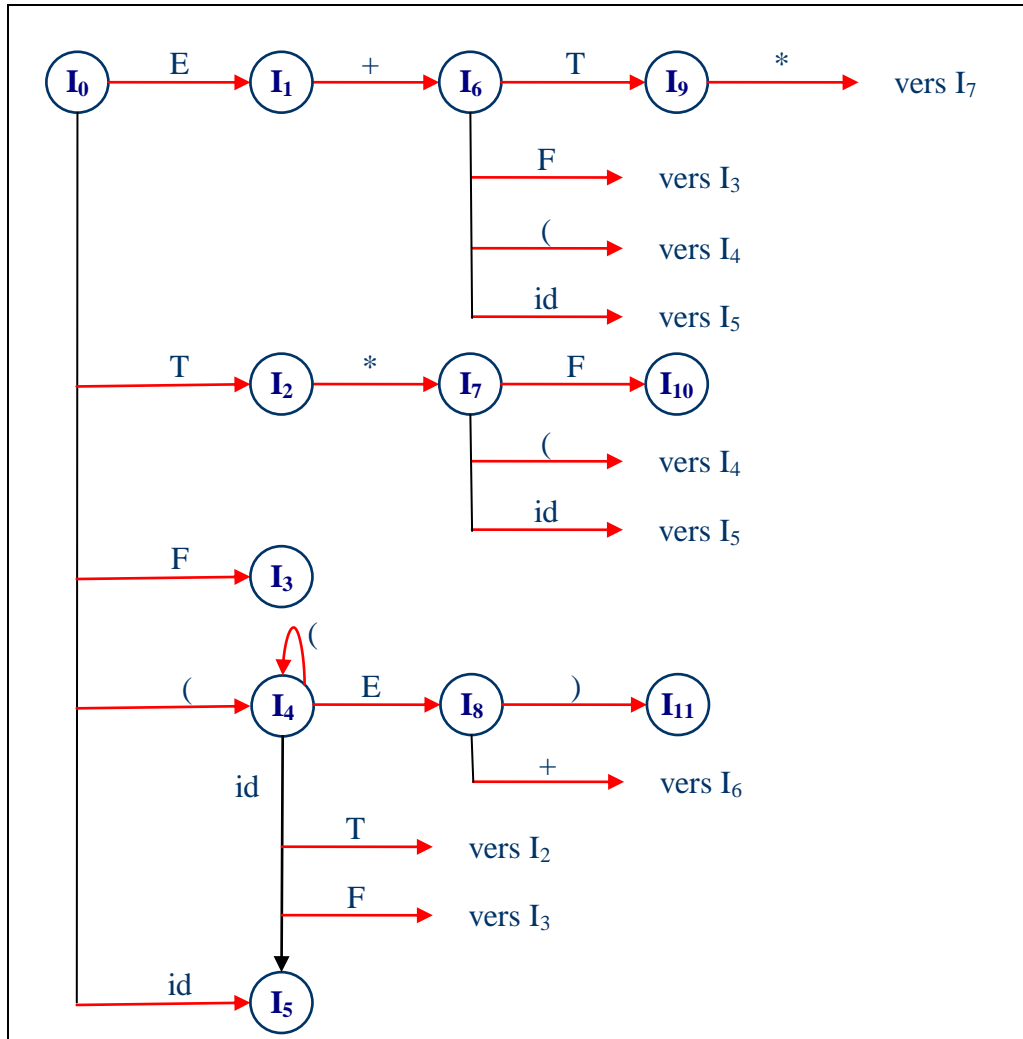


Figure 5.3. Représentation de l'AFD correspondant à l'exemple

### 5.3.3 Algorithme de construction de table d'analyse SLR

La construction d'une table d'analyse SLR pour une grammaire  $G$  est effectuée de la manière suivante :

- Effectuer une augmentation de  $G$  pour obtenir la grammaire augmentée  $G'$
- Construire  $C$  la collection canonique d'ensembles d'items LR(0) pour  $G'$ , soit  $C = \{I_0, I_1, I_2, \dots, I_N\}$
- Construire les fonctions ACTION (actions d'analyse) et SUCCESEUR (transition) à partir de  $C$  en utilisant l'algorithme suivant :

**Algorithme** ConstructionTableSLR;

**Début**

1/ L'état  $i$  est construit à partir de  $I_i$ . La partie ACTION pour l'état  $i$  est déterminée comme suit:

- Si**  $[A \rightarrow \alpha.a\beta] \in I_i$  et  $\text{Trans}(I_i, a) = I_j$  (avec  $a \in V_T$ ) **Alors** ACTION  $[i, a] \leftarrow d_j$  (décaler  $j$ )
- Si**  $[A \rightarrow \alpha.] \in I_i$  (avec  $A \neq S'$ ) **Alors** ACTION  $[i, a] \leftarrow r_{\text{num}}$  pour tout  $a \in \text{Suiv}(A)$   
(réduire par la règle  $A \rightarrow \alpha$  dont le numéro est num)
- Si**  $[S' \rightarrow S.] \in I_i$  **Alors** ACTION  $[i, \$] \leftarrow \text{accepter}$

2/ La partie SUCCESEUR pour l'état  $i$  est déterminée comme suit :

- Si**  $\text{Trans}(I_i, A) = I_j$  **Alors** SUCCESEUR  $[i, A] \leftarrow j$

3/ Toutes les entrées restantes dans la table sont mises à ERREUR

4/ L'état initial de l'analyseur est construit à partir de l'ensemble d'items contenant  $[S' \rightarrow S]$

**Fin ;**

### Exemple

En appliquant l'algorithme ci-dessus, on se propose de construire la table SLR correspondant à la grammaire G de l'exemple précédent (dont on a déterminé la collection canonique d'ensembles d'items). On obtient la table SLR suivante :

Etat	Action						Successeur		
	+	*	(	)	id	\$	E	T	F
0			d <sub>4</sub>		d <sub>5</sub>		1	2	3
1	d <sub>6</sub>					acc			
2	r <sub>2</sub>	d <sub>7</sub>		r <sub>2</sub>		r <sub>2</sub>			
3	r <sub>4</sub>	r <sub>4</sub>		r <sub>4</sub>		r <sub>4</sub>			
4			d <sub>4</sub>		d <sub>5</sub>		8	2	3
5	r <sub>6</sub>	r <sub>6</sub>		r <sub>6</sub>		r <sub>6</sub>			
6			d <sub>4</sub>		d <sub>5</sub>			9	3
7			d <sub>4</sub>		d <sub>5</sub>				10
8	d <sub>6</sub>			d <sub>11</sub>					
9	r <sub>1</sub>	d <sub>7</sub>		r <sub>1</sub>		r <sub>1</sub>			
10	r <sub>3</sub>	r <sub>3</sub>		r <sub>3</sub>		r <sub>3</sub>			
11	r <sub>5</sub>	r <sub>5</sub>		r <sub>5</sub>		r <sub>5</sub>			

Sachant que :

Suiv(E) = {+, ), \$}

Suiv(T) = {\*, +, ), \$}

Suiv(F) = {\*, +, ), \$}

### 5.3.4 Grammaire SLR (1)

Les tables obtenues avec l'algorithme défini dans la section précédente (5.3.3) sont appelées les **tables SLR(1)**. Un analyseur utilisant ces tables est appelé **analyseur SLR(1)**. Une grammaire est dite SLR(1) ou SLR si elle peut être représentée par une table SLR(1) dont chaque entrée est définie de façon unique (d /r /acc /erreur).

### Exemple

Soit la grammaire G, ayant les règles de production suivantes :

$S \rightarrow G=D \mid D$

$G \rightarrow *D \mid id$

$D \rightarrow G$

On se propose de montrer si cette grammaire est SLR(1).

On commence d'abord par augmenter la grammaire :

$S' \rightarrow S$

$S \rightarrow G=D$  ①

$S \rightarrow D$  ②

$G \rightarrow *D$  ③

$G \rightarrow id$  ④

$D \rightarrow G$  ⑤

Détermination de la collection canonique d'ensembles d'items :

$I_0 = \text{Ferm } \{[S' \rightarrow .S]\}$

=  $S' \rightarrow .S$

$S \rightarrow .G=D$

$S \rightarrow .D$

$G \rightarrow .*D$

$G \rightarrow .id$

$D \rightarrow .G$

$$I_1 = \text{Trans}(I_0, S) = \text{Ferm}\{[S' \rightarrow S.]\}$$

$$= S' \rightarrow S.$$

$$I_2 = \text{Trans}(I_0, G) = \text{Ferm}\{[S \rightarrow G.=D], [D \rightarrow G.]\}$$

$$= S \rightarrow G. =D$$

$$D \rightarrow G.$$

$$I_3 = \text{Trans}(I_0, D) = \text{Ferm}\{[S \rightarrow D.]\}$$

$$= S \rightarrow D.$$

$$I_4 = \text{Trans}(I_0, *) = \text{Ferm}\{[G \rightarrow *.D]\}$$

$$= G \rightarrow *.D$$

$$D \rightarrow .G$$

$$G \rightarrow *.D$$

$$G \rightarrow .id$$

$$I_5 = \text{Trans}(I_0, id) = \text{Ferm}\{[G \rightarrow id.]\}$$

$$= G \rightarrow id.$$

$$I_6 = \text{Trans}(I_2, =) = \text{Ferm}\{[S \rightarrow G.=D]\}$$

$$= S \rightarrow G.=D$$

$$D \rightarrow .G$$

$$G \rightarrow *.D$$

$$G \rightarrow .id$$

$$I_7 = \text{Trans}(I_4, D) = \text{Ferm}\{[G \rightarrow *.D.]\}$$

$$= G \rightarrow *.D.$$

$$I_8 = \text{Trans}(I_4, G) = \text{Ferm}\{[D \rightarrow G.]\}$$

$$= D \rightarrow G.$$

$$I_4 = \text{Trans}(I_4, *) = \text{Ferm}\{[G \rightarrow *.D]\}$$

$$= G \rightarrow *.D$$

$$I_5 = \text{Trans}(I_4, id) = \text{Ferm}\{[G \rightarrow id.]\}$$

$$I_9 = \text{Trans}(I_6, D) = \text{Ferm}\{[S \rightarrow G=D.]\}$$

$$= S \rightarrow G=D.$$

$$I_8 = \text{Trans}(I_6, G) = \text{Ferm}\{[D \rightarrow G.]\}$$

$$= D \rightarrow G.$$

$$I_4 = \text{Trans}(I_6, *) = \text{Ferm}\{[G \rightarrow *.D]\}$$

$$I_5 = \text{Trans}(I_6, id) = \text{Ferm}\{[G \rightarrow id.]\}$$

$$= G \rightarrow id.$$

Construction de la table d'analyse SLR :

Etat	Action				Successeur		
	=	*	id	\$	S	G	D
0		d <sub>4</sub>	d <sub>5</sub>		1	2	3
1				acc			
2	d <sub>6</sub> /r <sub>5</sub>			r <sub>5</sub>			
3				r <sub>2</sub>			
4		d <sub>4</sub>	d <sub>5</sub>			8	7
5	r <sub>4</sub>			r <sub>4</sub>			
6		d <sub>4</sub>	d <sub>5</sub>			8	9
7	r <sub>3</sub>			r <sub>3</sub>			
8	r <sub>5</sub>			r <sub>5</sub>			
9				r <sub>1</sub>			

Sachant que :

Suiv(D) = {=, \$}

Suiv(G) = {=, \$}

Suiv(S) = { \$}

### Remarque

On constate que Action[2,=] est une case définie de façon multiple. L'état 2 présente un conflit d/r (déclarer/réduire) ou s/r (shift/reduce) sur le symbole d'entrée « = » donc la grammaire **n'est pas SLR (1)**.

Dans notre cas, la grammaire n'est pas ambiguë mais elle n'est pas SLR(1). D'ailleurs, beaucoup de grammaires non ambiguës ne sont pas SLR(1). Le conflit shift/réduire (décaler/réduire) dans l'exemple précédent provient du manque de puissance de la méthode SLR qui ne peut pas « se rappeler » assez de contexte gauche pour décider de l'action de l'analyseur sur l'entrée « = » après avoir « vu » une chaîne pouvant être réduite vers G.

**Les méthodes LR canonique et LALR réussissent pour un nombre plus important de grammaires que la méthode SLR.**

## 5.4 Construction de tables d'analyse LR canonique ou LR(1)

La construction de tables d'analyse LR canonique ou LR(1) est la technique la plus générale de construction de tables d'analyse LR pour une grammaire. Cette méthode permet d'attacher plus d'informations à un état pour éviter un certain nombre d'actions invalides et donc de conflits.

### 5.4.1 Items LR(1)

La forme générale d'un item LR(1) est  $[A \rightarrow \alpha.\beta, a]$  sachant que :

$A \rightarrow \alpha\beta$  est une règle de production de la grammaire et  $a \in V_T \cup \{\$, \epsilon\}$ .

Le 1 du LR(1) fait référence à la longueur du second composant, appelé **symbole de pré-vision** de l'item. La prévision n'a aucun effet dans un item de la forme  $[A \rightarrow \alpha.\beta, a]$  avec  $\beta \neq \epsilon$ , mais un item de la forme  $[A \rightarrow \alpha., a]$  implique « réduire par  $A \rightarrow \alpha$  uniquement lorsque le prochain symbole d'entrée est  $a$  ». Ceci signifie que la réduction par  $A \rightarrow \alpha$  ne se fait que sur les symboles d'entrée  $a$  pour lesquels  $[A \rightarrow \alpha., a]$  est un item LR(1) de l'état en sommet de pile. L'ensemble de tels  $a$  sera toujours un sous-ensemble de  $\text{Suiv}(A)$ .

### 5.4.2 Fermeture d'ensemble d'items LR(1)

La fonction  $\text{Ferm}(I)$  peut être définie de la manière suivante :

<b>Fonction</b> $\text{Ferm}(I);$ <b>Début</b> <b>Répéter</b> <b>Pour</b> chaque item $[A \rightarrow \alpha.B\beta, a] \in I$ , chaque production $B \rightarrow \delta \in G'$ et chaque terminal $b \in \text{Prem}(\beta a)$ tel que $[B \rightarrow \delta, b] \notin I$ <b>Faire</b> Ajouter $[B \rightarrow \delta, b]$ à $I$ <b>Jusqu'à</b> ce qu'aucun item ne puisse être ajouté à $I$ ; $\text{Ferm}(I) := I;$ <b>Fin ;</b>
--

### 5.4.3 Fonction Transition

Soit  $I$  un ensemble d'items LR(1) et  $X$  un symbole de la grammaire ( $X \in V_T \cup V_N$ ). Transition  $(I, X)$  est définie de la manière suivante :

<b>Fonction</b> $\text{Trans}(I, X);$ <b>Début</b> Soit $J$ l'ensemble des items $[A \rightarrow \alpha X.\beta, a]$ tels que $[A \rightarrow \alpha.X\beta, a] \in I$ . $\text{Trans}(I, X) := \text{Ferm}(J);$ <b>Fin ;</b>
---



#### 5.4.4 Construction de la collection d'ensembles d'items LR(1)

L'algorithme suivant permet de déterminer la collection d'ensemble d'items LR(1) pour une grammaire augmentée  $G'$

**Procédure** CollectionEnsembleItems;

**Début**

$C := \{\text{Ferm}(\{[S' \rightarrow .S, \$]\})\}$  ;

**Répéter**

**Pour** chaque ensemble d'items  $I$  de  $C$  et pour chaque symbole de la grammaire  $X$  tel que  $\text{Trans}(I, X)$  soit non vide et non encore dans  $C$

**Faire** ajouter  $\text{Trans}(I, X)$  à  $C$

**Jusqu'à** ce qu'aucun ensemble d'items ne puisse être ajouté à  $C$ ;

**Fin** ;

#### 5.4.5 Algorithme de construction de tables d'analyse LR(1)

La construction d'une table d'analyse LR(1) pour une grammaire  $G$  est effectuée de la manière suivante :

- Effectuer une augmentation de  $G$  pour obtenir la grammaire augmentée  $G'$ .
- Construire  $C$  la collection d'ensembles d'items LR(1) pour  $G'$ .
- Construire les fonctions ACTION (actions d'analyse) et SUCCESSEUR (transition) à partir de  $C$  en utilisant l'algorithme suivant :

**Algorithme** ConstructionTableLR;

**Début**

1/ L'état  $i$  est construit à partir de  $I_i$ . La partie ACTION pour l'état  $i$  est déterminée comme suit:

a. **Si**  $[A \rightarrow \alpha.a\beta, b] \in I_i$  et  $\text{Trans}(I_i, a) = I_j$  (avec  $a \in V_T$ ) **Alors** ACTION  $[i, a] \leftarrow d_j$  (décaler  $j$ )

b. **Si**  $[A \rightarrow \alpha., a] \in I_i$  (avec  $A \neq S'$ ) **Alors** ACTION  $[i, a] \leftarrow r_{\text{num}}$   
(réduire par la règle  $A \rightarrow \alpha$  dont le numéro est num )

c. **Si**  $[S' \rightarrow .S, \$] \in I_i$  **Alors** ACTION  $[i, \$] \leftarrow \text{accepter}$

2/ La partie SUCCESSEUR pour l'état  $i$  est déterminée comme suit :

**Si**  $\text{Trans}(I_i, A) = I_j$  **Alors** SUCCESSEUR  $[i, A] \leftarrow j$

3/ Toutes les entrées restantes dans la table sont mises à ERREUR

4/ L'état initial de l'analyseur est construit à partir de l'ensemble d'items contenant  $[S' \rightarrow .S, \$]$

**Fin** ;

#### 5.4.6 Grammaire LR(1)

Les tables obtenues avec l'algorithme défini dans la section précédente (5.4.5) sont appelées les **tables canoniques d'analyse LR(1)**. Un analyseur utilisant ces tables est appelé **analyseur LR(1) canonique**. Une grammaire est dite LR(1) ou LR si elle peut être représentée par une table LR(1) dont chaque entrée est définie de façon unique.

**Toute grammaire SLR(1) est une grammaire LR(1), mais, pour une grammaire SLR(1), l'analyseur LR canonique peut avoir un nombre d'états supérieur à l'analyseur SLR pour la même grammaire.**

#### 5.4.7 Exemple de construction de tables d'analyse LR(1)

Soit la grammaire  $G$ , ayant les règles de production suivantes :

$S \rightarrow CC$

$C \rightarrow cC \mid d$

On se propose de montrer si cette grammaire LR(1) (Question 1 de l'exercice 6 de la série de TD no : 4).

- On commence d'abord par augmenter la grammaire :

$S' \rightarrow S$

$S \rightarrow CC$  ①

$C \rightarrow cC$  ②

$C \rightarrow d$  ③

- Détermination de la collection d'ensembles d'items LR(1) :

$I_0 = \text{Ferm} \{[S' \rightarrow \cdot S, \$]\}$

$= S' \rightarrow \cdot S, \$$   $\text{Prem}(\beta a) = \text{Prem}(\$) = \{\$ \}$

$S \rightarrow \cdot CC, \$$   $\text{Prem}(\beta a) = \text{Prem}(C\$) = \{c, d\}$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

$I_1 = \text{Trans}(I_0, S) = \text{Ferm}\{[S' \rightarrow S \cdot, \$]\}$

$= S' \rightarrow S \cdot, \$$

$I_2 = \text{Trans}(I_0, C) = \text{Ferm}\{[S \rightarrow \cdot C.C, \$]\}$

$= S \rightarrow \cdot C.C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

$I_3 = \text{Trans}(I_0, c) = \text{Ferm}\{[C \rightarrow \cdot c.C, c/d]\}$

$= C \rightarrow \cdot c.C, c/d$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

$I_4 = \text{Trans}(I_0, d) = \text{Ferm}\{[C \rightarrow \cdot d, c/d]\}$

$= C \rightarrow \cdot d, c/d$

$I_5 = \text{Trans}(I_2, C) = \text{Ferm}\{[S \rightarrow CC \cdot, \$]\}$

$= S \rightarrow CC \cdot, \$$

$I_6 = \text{Trans}(I_2, c) = \text{Ferm}\{[C \rightarrow c \cdot C, \$]\}$

$= C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

$I_7 = \text{Trans}(I_2, d) = \text{Ferm}\{[C \rightarrow d \cdot, \$]\}$

$= C \rightarrow d \cdot, \$$

$I_8 = \text{Trans}(I_3, C) = \text{Ferm}\{[C \rightarrow cC \cdot, c/d]\}$

$= C \rightarrow cC \cdot, c/d$

$I_9 = \text{Trans}(I_3, c) = \text{Ferm}\{[C \rightarrow c \cdot C, c/d]\}$

$I_{10} = \text{Trans}(I_3, d) = \text{Ferm}\{[C \rightarrow d \cdot, c/d]\}$

$I_{11} = \text{Trans}(I_6, C) = \text{Ferm}\{[C \rightarrow cC \cdot, \$]\}$

$= C \rightarrow cC \cdot, \$$

$I_{12} = \text{Trans}(I_6, c) = \text{Ferm}\{[C \rightarrow c \cdot C, \$]\}$

$I_{13} = \text{Trans}(I_6, d) = \text{Ferm}\{[C \rightarrow d \cdot, \$]\}$

- Construction de la table d'analyse LR :

Etat	Action			Successeur	
	c	d	\$	S	C
0	d <sub>3</sub>	d <sub>4</sub>		1	2
1			acc		
2	d <sub>6</sub>	d <sub>7</sub>			5
3	d <sub>3</sub>	d <sub>4</sub>			8
4	r <sub>3</sub>	r <sub>3</sub>			
5			r <sub>1</sub>		
6	d <sub>6</sub>	d <sub>7</sub>			9
7			r <sub>3</sub>		
8	r <sub>2</sub>	r <sub>2</sub>			
9			r <sub>2</sub>		

## 5.5 Construction de tables d'analyse LALR

La méthode LALR(1) (Look Ahead LR) est souvent utilisée en pratique car c'est une méthode ayant un coût et une efficacité intermédiaire en comparaison avec SLR(1) et LR(1). Pour une grammaire donnée, la table d'analyse LALR(1) occupe autant de place (même nombre d'états) que la table SLR(1) (moins que la table LR(1)) et satisfait une grande classe de grammaires. Ainsi, il est plus intéressant, plus facile et plus économique de construire des tables SLR ou LALR que des tables LR canoniques. A titre d'exemple, les méthodes SLR et LALR donnent un nombre d'états de plusieurs centaines pour un langage tel que Pascal alors qu'on arrive à plusieurs milliers d'états avec la méthode LR canonique.

### 5.5.1 Algorithme de construction de tables LALR(1)

Cet algorithme se base sur la construction de la collection d'ensembles d'items LR(1) ainsi que la table d'analyse LR(1). Il utilise la notion de cœur d'item LR(1), sachant que : **item LR(1)=[cœur, symbole de pré-vision]**.

L'idée générale de cet algorithme est de construire les ensembles d'items LR(1) et de fusionner les ensembles ayant un cœur commun. La table d'analyse LALR(1) est construite à partir de la collection des ensembles d'items fusionnés. Signalons que cet algorithme est le plus simple, mais il existe d'autres algorithmes plus efficaces pour la construction de tables LALR(1), qui ne se basent pas sur la méthode LR(1).

**Algorithme** ConstructionTableLALR;

**Début**

1/ Construire la collection d'ensembles d'items LR(1) pour la grammaire augmentée  $G'$ .

2/ Pour chaque cœur présent parmi les ensembles d'items LR(1), trouver tous les états ayant ce même cœur et remplacer ces états par leur union.

3/ La table LALR(1) est obtenue en condensant la table LR(1) par superposition des lignes correspondant aux états regroupés.

**Fin ;**

Pendant la superposition des lignes, il y a un risque de conflit :

**Cas 1** : Conflit décaler/décaler ou shift/shift ( $d_i/d_j$ )

Ce cas de conflit ne peut pas se produire car le décalage se base sur l'élément après le point.

**Cas 2** : Conflit décaler/réduire ou shift/reduce ( $d/r$ )

Ce cas de conflit ne peut se produire que s'il existe dans la table LR(1) pour l'un, au moins des états d'origine.

**Cas 3** : Conflit réduire/réduire ou reduce/reduce ( $r_i/r_j$ )

C'est le seul conflit possible. Il se produit quand deux items LR(1) ayant respectivement la  $[A \rightarrow C., x]$  et  $[B \rightarrow C., x]$  appartiennent à un ensemble obtenu après regroupement d'états.

**Conclusion**: Pendant la superposition des lignes, le seul cas de conflit qui peut se produire est réduire/réduire.

**Remarque**

S'il n'y a pas de conflit dans la table LALR(1), la grammaire sera considérée LALR(1) ou LALR.

### 5.5.2 Exemple de construction de tables d'analyse LALR(1)

On se propose de construire la table d'analyse LALR(1) pour la grammaire  $G$ , traitée dans la section 5.4.7 (Question 2 de l'exercice 6 de la série de TD no :4):

$S \rightarrow CC$

$C \rightarrow cC \mid d$

Pour la détermination de la collection d'ensembles d'items LALR(1), on recherche les ensembles d'items LR(1) pouvant être fusionnés, on en trouve trois paires :  $I_3$  avec  $I_6$ ,  $I_4$  avec  $I_7$  et  $I_8$  avec  $I_9$  qui seront remplacés par leurs unions respectives :

$I_{36} = C \rightarrow c.C, c/d/\$$

$C \rightarrow .cC, c/d/\$$

$C \rightarrow .d, c/d/\$$

$I_{47} = C \rightarrow d., c/d/\$$

$I_{89} = C \rightarrow cC., c/d/\$$

Les autres ensembles d'items (non concernés par les fusions) restent inchangés.

La construction de la table d'analyse LALR est obtenue par superposition des lignes correspondant aux états fusionnés:

Etat	Action			Successeur	
	c	d	\$	S	C
0	$d_{36}$	$d_{47}$		1	2
1			acc		
2	$d_{36}$	$d_{47}$			5
36	$d_{36}$	$d_{47}$			89
47	$r_3$	$r_3$	$r_3$		
5			$r_1$		
89	$r_2$	$r_2$	$r_2$		

### 5.5.3 Analyse de chaînes par la méthode LALR(1)

Si la chaîne analysée est correcte syntaxiquement (appartient au langage), l'analyse LALR(1) progressera de la même manière que LR(1). Les seules différences sont dans l'appellation ou la numérotation des états de transition.

Si la chaîne analysée est erronée syntaxiquement (n'appartient pas au langage), l'erreur sera détectée plus rapidement par l'analyseur LR(1) alors que l'analyseur LALR(1) peut effectuer plusieurs réductions avant de signaler l'erreur.

*Ces deux cas sont illustrés à travers la réponse aux questions 4 et 5 de l'exercice 6.*

## SERIE DE TD N°:4 COMPILATION, ANALYSE SYNTAXIQUE ASCENDANTE

### Exercice 1

Soit la grammaire G ayant les règles de production suivantes :

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

1. Cette grammaire est-elle une grammaire SLR?
2. Analyser la chaîne ab en utilisant la table d'analyse SLR et en résolvant les conflits éventuels.

### Exercice 2

Soit la grammaire augmentée ayant les règles de production suivantes :

$S \rightarrow E$

$E \rightarrow E+E \mid E * E \mid nb$

1. Construire la table d'analyse syntaxique SLR pour cette grammaire.
2. Utiliser cette table pour analyser les chaînes  $1+2+3$  et  $1+2*3$  en résolvant les conflits éventuels.

### Exercice 3

Soit la grammaire ayant les règles de production suivantes :

$S \rightarrow \text{if } a \text{ then } S \mid \text{if } a \text{ then } S \text{ else } S \mid i$

1. Construire l'arbre de dérivation de la chaîne **if a then if a then i else i**. Que peut-on conclure ?
2. Cette grammaire peut-elle être une grammaire SLR ? Justifier votre réponse.
3. Construire la table d'analyse SLR pour cette grammaire.
4. Utiliser cette table pour analyser la chaîne **if a then if a then i else i** en résolvant les conflits éventuels.

### Exercice 4 (à faire à la maison, pas de correction en TD, quelques indications de réponses dans l'énoncé de l'exercice)

Soit la grammaire  $G = (\{\text{debut, fin, }, ;, \text{ si, alors, sinon, a, :=, et }\}, \{A, L, I, E\}, A, P)$  où P est défini par les règles de production suivantes :

$A \rightarrow \text{debut } L \text{ fin } .$

$L \rightarrow L ; I \mid I$

$I \rightarrow \text{si } E \text{ alors } I \mid \text{si } E \text{ alors } I \text{ sinon } I \mid a := E$

$E \rightarrow E \text{ et } E \mid a$

1. Construire la collection d'ensembles d'items LR(0) pour cette grammaire.  
(Réponse partielle : on trouve 21 ensembles d'items de  $I_0$  jusqu'à  $I_{20}$ )
2. Dresser la table d'analyse syntaxique SLR. (Réponse partielle : on trouve 2 conflits d/r)
3. Utiliser cette table pour analyser la chaîne : **debut si a et a alors a := a sinon a := a fin.** puis déduire son arbre de dérivation. (Réponse partielle : la chaîne est acceptée après 26 étapes d'analyse)

### Exercice 5 (correction en cours)

Soit la grammaire G ayant les règles de production suivantes :

$S \rightarrow G=D \mid D$

$G \rightarrow *D \mid id$

$D \rightarrow G$

1. Cette grammaire est-elle une grammaire LR(1)?
2. Utiliser la table LR(1) pour analyser les chaînes : **id=id** et **id==id** puis déduire leurs arbres de dérivation respectifs.

### Exercice 6 (correction en cours)

Soit la grammaire G ayant les règles de production suivantes :

$S \rightarrow CC$

$C \rightarrow cC \mid d$

1. Cette grammaire est-elle une grammaire LR(1)?
2. Cette grammaire est-elle une grammaire LALR(1)?
3. Peut-on déduire si cette grammaire est SLR ? Quelle est la taille de sa table SLR ?
4. Analyser la chaîne ccdccd et déduire son arbre de dérivation en utilisant les méthodes LR et LALR. Conclusion ?
5. Analyser la chaîne ccd et déduire son arbre de dérivation en utilisant les méthodes LR et LALR. Conclusion ?

### Exercice 7 (correction en cours)

Soit la grammaire G ayant les règles de production suivantes :

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

1. Cette grammaire est-elle une grammaire LR(1)? Utiliser la table LR(1) pour analyser les chaînes **bcd** et **bcbd**.
2. Peut-on déduire si cette grammaire est une grammaire LALR(1)?
3. Construire la table LALR(1).

## **CHAPITRE 6 : TRADUCTION DIRIGEE PAR LA SYNTAXE ET ANALYSE SEMANTIQUE**

---

### **6.1 Introduction**

---

La grammaire hors contexte d'un langage de programmation ne peut pas décrire certaines propriétés fondamentales qui dépendent du contexte telles que :

- L'impossibilité d'utiliser dans ses instructions une variable non déclarée préalablement
- L'impossibilité de déclarer deux fois la même variable
- L'impossibilité de multiplier un réel par une chaîne de caractères
- La nécessité de correspondance (en nombre et en type) entre les paramètres formels et les paramètres effectifs des procédures et des fonctions

L'analyse sémantique (contextuelle) a pour rôle de vérifier ce genre de contraintes. Elle se fait, en général, en associant des règles sémantiques aux productions de la grammaire. Deux notations peuvent être utilisées pour effectuer cette association :

- Les définitions dirigées par la syntaxe ou
- Les schémas de traduction

Les règles sémantiques calculent la valeur des attributs attachés au symbole de la grammaire. Conceptuellement la chaîne en entrée est analysée syntaxiquement pour construire un arbre qui est parcouru autant de fois que nécessaire pour évaluer les règles sémantiques à ses nœuds. L'évaluation des règles sémantiques peut produire du code, sauvegarder de l'information dans la table des symboles, etc., pour obtenir la traduction de la chaîne en entrée.

### **6.2 Définitions dirigées par la syntaxe (DDS)**

---

#### **6.2.1 Introduction**

---

Une DDS est un formalisme permettant d'associer des actions à une règle de production de la grammaire. On appelle ainsi DDS la donnée d'une grammaire et des règles sémantiques. On parle de grammaire **attribuée**.

Dans une DDS, on a les éléments suivants :

- Chaque symbole de la grammaire ( $\in V_T$  ou  $V_N$ ) possède un ensemble d'**attributs** (variables) pouvant être **synthétisés** ou **hérités** de ce symbole. Si un nœud d'arbre syntaxique (étiqueté par un symbole) est représenté par un enregistrement, un attribut correspond à un nom de champ de cet enregistrement.
- Chaque règle de production de la grammaire possède un ensemble de règles sémantiques qui permettent de calculer la valeur des attributs associés au symbole apparaissant dans la production
- Une règle sémantique est une suite d'instructions algorithmiques (elle peut contenir des affectations, des instructions d'affichage, des instructions de contrôle, etc.)

Les règles sémantiques impliquent des dépendances entre les attributs. Ces dernières sont représentées par un graphe de dépendance (GD) à partir duquel on déduit un ordre d'évaluation des règles sémantiques.

## Remarque

Un arbre syntaxique muni des valeurs d'attributs à chaque nœud est appelé **arbre décoré** (ou **annoté**). Le processus de calcul des valeurs des attributs aux nœuds est appelé décoration (ou annotation) de l'arbre.

### 6.2.2 Structure d'une DDS

Dans une DDS, chaque production de la grammaire ( $A \rightarrow \alpha$ ) possède un ensemble de règles sémantiques de la forme :

$b := f(C1, C2, \dots, Ck)$

$f$  : est une fonction, souvent écrite sous forme d'expressions et parfois sous forme d'un appel de procédure.

$b$  : est soit un attribut synthétisé de  $A$ , soit un attribut hérité d'un symbole en partie droite de la règle.

$C1, C2, \dots, Ck$  : sont des attributs de symboles quelconques dans la règle de production.

### 6.2.3 Attributs synthétisés

La valeur d'un attribut synthétisé en un nœud est calculée à partir des attributs au niveau des fils de ce nœud dans l'arbre syntaxique.

Les attributs synthétisés sont très utilisés en pratique. Une DDS qui utilise uniquement les attributs synthétisés est appelée *Définition S-attribuée*.

Un arbre syntaxique pour une définition S-attribuée est toujours décoré en évaluant les règles sémantiques pour les attributs de chaque nœud du bas vers le haut (peut être adapté à une analyse ascendante : grammaire LR, par exemple).

## Exemple

Soit la grammaire d'un programme de calculatrice de bureau :

$L \rightarrow En$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{Chiffre}$

Productions	Règles sémantiques
$L \rightarrow En$	Imprimer ( $E.val$ )
$E \rightarrow E1 + T$	$E.val := E1.val + T$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T1 * F$	$T.val := T1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{Chiffre}$	$F.val := \text{Chiffre.valLex}$

## Remarques

- Cette DDS associe des attributs synthétisés « val » de type entier à chacun des non terminaux  $E$ ,  $T$  et  $F$ .
- Le terminal Chiffre a un attribut synthétisé valLex (valeur lexicale) dont la valeur est fournie par l'analyse lexicale.
- Imprimer( $E.val$ ) est un appel de procédure pour imprimer la valeur de l'expression arithmétique  $E$ .

## Exemple

Soit la chaîne  $3*4+5n$ . On veut établir l'arbre syntaxique correspondant à cette chaîne, avant et après la décoration, en utilisant la DDS précédente.

Avant la décoration, on obtient l'arbre de la figure 6.1.

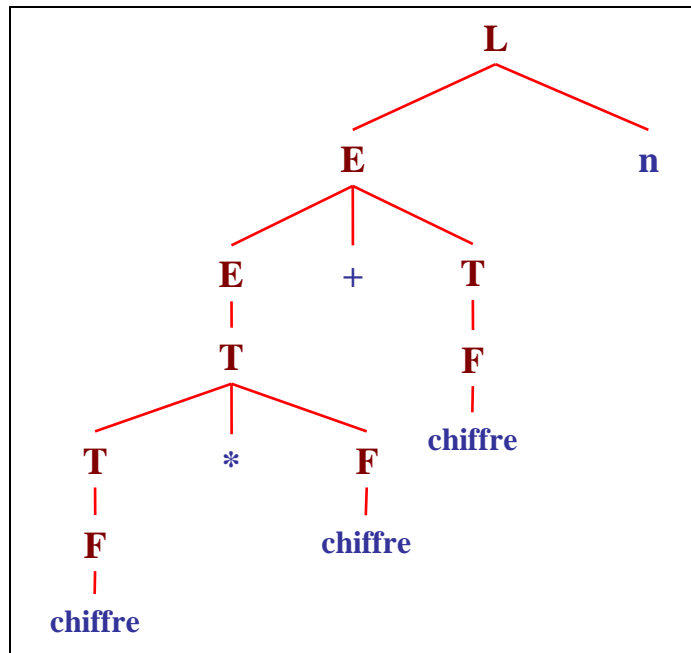


Figure 6.1. Arbre syntaxique de l'expression 3\*4+5

Après la décoration, nous obtenons l'arbre décoré de la figure 6.2.

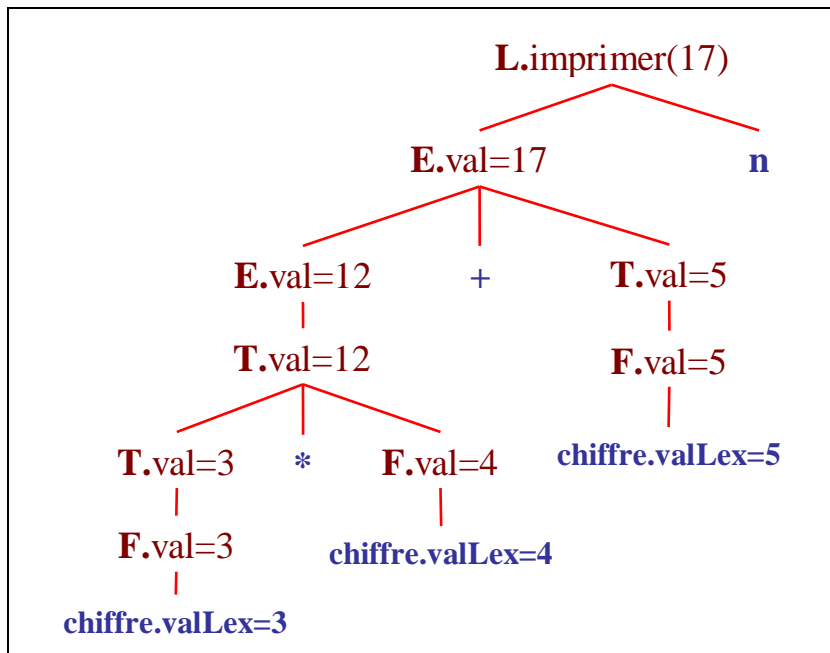


Figure 6.2. Arbre syntaxique décoré de l'expression 3\*4+5

### Remarque

Dans une DDS, les terminaux de la grammaire ne sont sensés avoir que des attributs synthétisés dont les valeurs sont fournies, généralement, par l'analyseur lexical.

### 6.2.4 Attributs hérités

Un attribut hérité est un attribut dont la valeur, à un nœud d'un arbre syntaxique, est définie en fonction du père et/ou des frères de ce nœud. Il est souvent plus naturel d'utiliser des attributs hérités qui sont bien adaptés à l'expression des dépendances contextuelles.

### Exemple

Soit une grammaire permettant de déclarer des entiers et des réels :



$D \rightarrow T L$   
 $T \rightarrow \text{entier} \mid \text{réel}$   
 $L \rightarrow L, \text{id} \mid \text{id}$

Dans la DDS suivante, on utilise un attribut hérité pour distribuer l'information de typage aux différents identificateurs d'une déclaration.

Productions	Règles sémantiques
$D \rightarrow T L$	$L.Type\ h := T.type$
$T \rightarrow \text{entier}$	$T.Type := \text{entier}$
$T \rightarrow \text{réel}$	$T.Type := \text{réel}$
$L \rightarrow L1, \text{id}$	$L1.Type\ h := L.Type\ h$ ajouterType(id.entrée, L.Type h)
$L \rightarrow \text{id}$	ajouterType (id.entrée, L.Type h)

La première règle sémantique  $L.Type\ h := T.Type$ , associée à la production  $D \rightarrow T L$ , donne à l'attribut hérité  $L.Type\ h$  la valeur de type de la déclaration (entier ou réel). Les autres règles sémantiques font descendre ce type le long de l'arbre par l'intermédiaire de l'attribut hérité  $L.Type\ h$ . La procédure ajouterType() est appelée pour permettre d'ajouter le type de chaque identificateur au niveau de son entrée dans la table des symboles.

### Exemple

En utilisant la DDS précédente, donner l'arbre syntaxique décoré de la déclaration : entier a, b, c

Avant la décoration, nous avons l'arbre syntaxique de la figure 6.3.

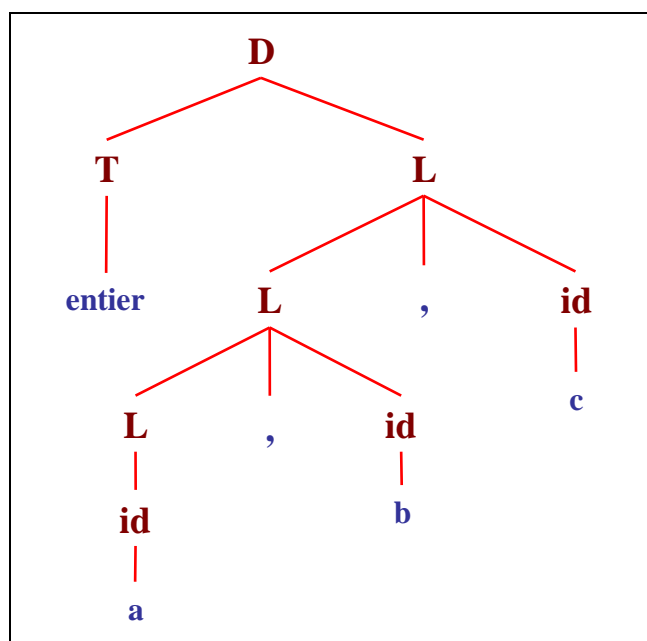


Figure 6.3. Arbre syntaxique de : entier a, b, c

Après la décoration, nous obtenons l'arbre syntaxique décoré de la figure 6.4.

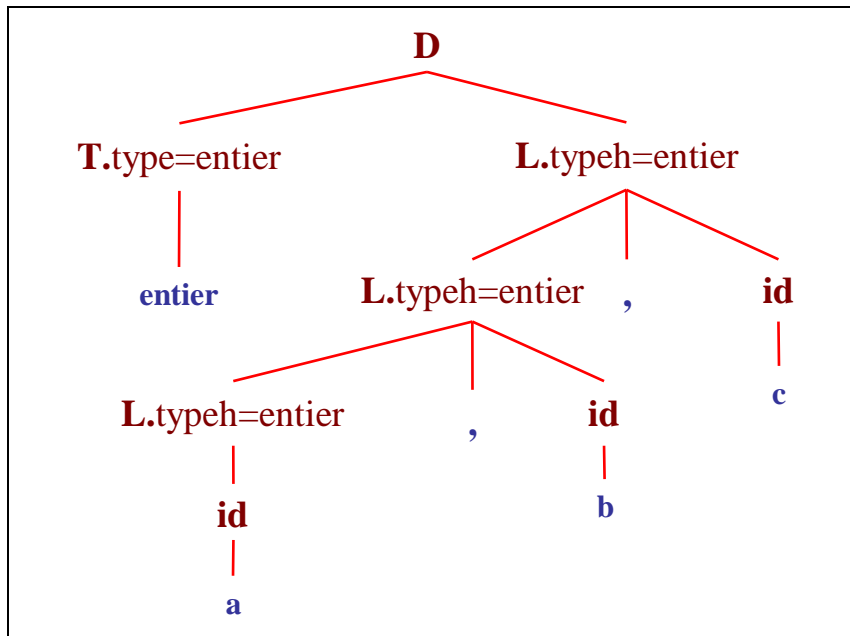


Figure 6.4. Arbre syntaxique décoré de : entier a, b, c

### 6.2.5 Graphe de dépendances

On appelle graphe de dépendance un graphe orienté représentant les interdépendances entre les attributs au niveau des nœuds un arbre syntaxique. Ce graphe permet de déduire l'ordre d'évaluation des règles sémantiques. Le graphe a un sommet pour chaque attribut. Si un attribut **b** dépend d'un attribut **c**, il y aura un arc du sommet c vers b, ceci signifie que la règle sémantique qui définit c doit être évaluée avant celle qui définit b.

#### Exemple 1

Productions	Règles sémantiques
$E \rightarrow E1 + E2$	$E.val := E1.val + E2.val$

A chaque fois qu'on utilise la production  $E \rightarrow E1 + E2$  dans un arbre syntaxique, on ajoute les deux arcs ci-dessous (voir figure 6.5) au niveau du graphe de dépendance pour indiquer que l'évaluation de E.val doit être effectuée après celle de E1.val et E2.val

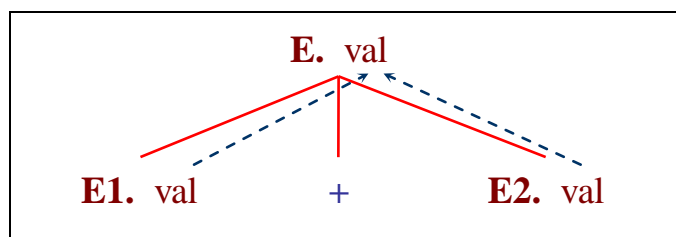


Figure 6.5. Graphe des dépendances de  $E \rightarrow E1 + E2$

#### Exemple 2

Productions	Règles sémantiques
$A \rightarrow XY$	$A.a := f(X.x, Y.y)$ $X.h := g(A.a, Y.y)$

Le graphe de dépendance est donné dans la figure 6.6.

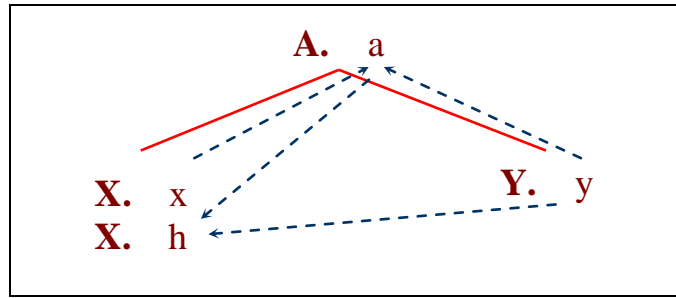


Figure 6.6. Graphe des dépendances de  $A \rightarrow XY$

Algorithme de construction du graphe des dépendances pour un arbre syntaxique donné

**Pour** chaque nœud  $n$  de l'arbre syntaxique **faire**

**Pour** chaque attribut  $a$  du symbole de la grammaire étiquetant  $n$  **faire**

        Construire un sommet dans le graphe des dépendances pour  $a$

**Pour** chaque nœud  $n$  de l'arbre syntaxique **faire**

**Pour** chaque règle sémantique  $b := f(c_1, \dots, c_k)$  associé à la production appliquée en  $n$  **faire**

**Pour**  $i := 1$  à  $k$  **faire** construire un arc du sommet associé à  $c_i$  au sommet associé à  $b$

## 6.3 Utilisation des DDS pour la construction de représentation graphiques associées aux langages

### 6.3.1 Arbre abstrait

Un arbre abstrait est une forme condensée d'arbre syntaxique adaptée à la représentation des constructions des langages. Dans un arbre abstrait, les opérateurs et les mots clés n'apparaissent pas comme des feuilles mais sont plutôt associés au nœud intérieur qui sera le père de cette feuille dans l'arbre. De plus, les chaînes de productions simples peuvent être éliminées.

La traduction dirigée par la syntaxe peut se baser sur les arbres abstraits, de la même façon que sur les arbres syntaxiques concrets, en attachant des attributs aux nœuds. L'utilisation des arbres abstraits comme représentation intermédiaire permet de dissocier la traduction de l'analyse syntaxique.

#### Exemple 1

Soit à représenter l'arbre syntaxique concret et l'arbre abstrait de la chaîne  $3 * 4 + 5$  en utilisant la grammaire :

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \text{Chiffre}$

La figure 6.7 illustre les deux arbres.

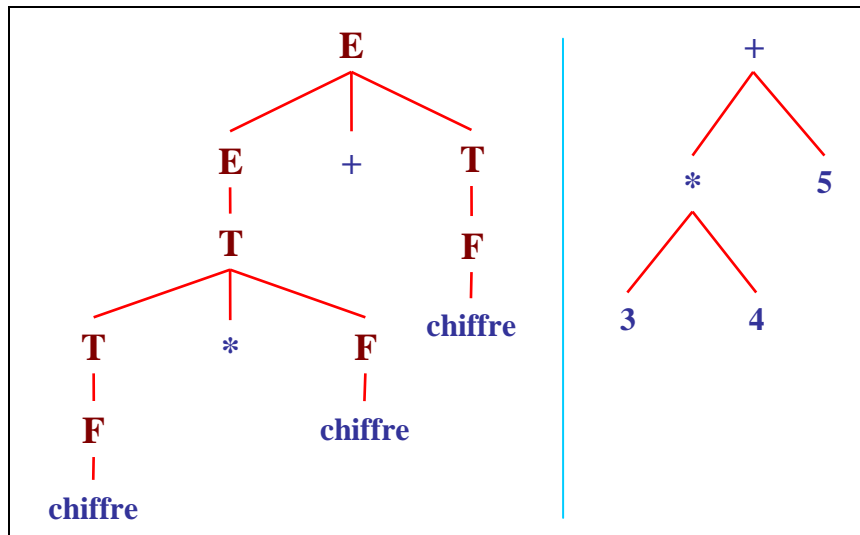


Figure 6.7. Arbre syntaxique concret et arbre abstrait de la chaîne 3 \* 4 + 5

## Exemple 2

Soit la règle de production  $I \rightarrow \text{Si } C \text{ alors } I1 \text{ Sinon } I2$

La figure 6.8 donne les deux arbres

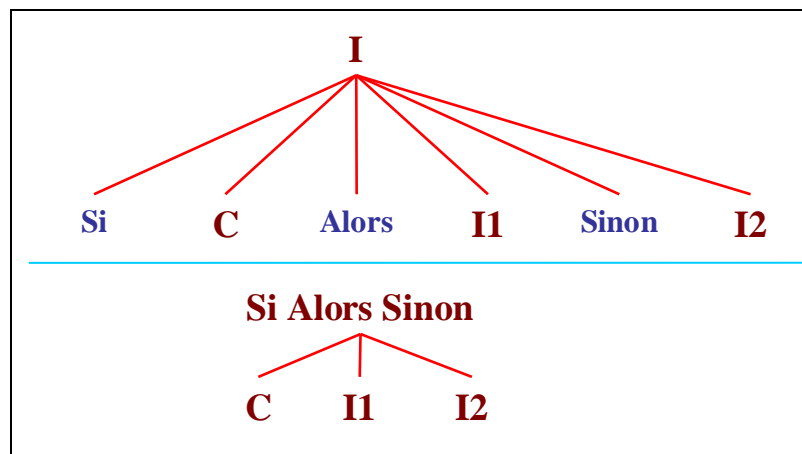


Figure 6.8. Arbre syntaxique concret et arbre abstrait de l'instruction Si-Alors-Sinon

### 6.3.2 Construction de l'arbre abstrait pour les expressions

La construction d'un arbre abstrait pour une expression est semblable à la traduction de l'expression en forme post-fixée (3 \* 4 + 5 en forme post-fixée sera 3 4 \* 5 +).

Chaque nœud de l'arbre abstrait peut être implémenté sous la forme suivante :

Opérateur	gauche	droit
-----------	--------	-------

Pour créer les nœuds des arbres abstraits correspondant à des expressions avec des opérateurs binaires, on utilise les fonctions suivantes :

- 1- CréerNoeud (Op, gauche, droit) : Crée un nœud opérateur ayant une étiquette op avec deux champs contenant des pointeurs vers gauche et droit.
- 2- CréerFeuille(id, entrée) : Crée un nœud identificateur (feuille) correspondant à un identificateur ayant pour étiquette id avec un champ entrée contenant un pointeur vers l'entrée de l'identificateur dans la table des symboles.

3- CréerFeuille(nb, val) : Crée une feuille associée à un nombre ayant pour étiquette « nb » avec un champ « val » contenant la valeur de ce nombre.

### Exemple

On veut déterminer la séquence d'appels pour créer l'arbre abstrait représentant l'expression  $a - 4 + c$

La séquence d'appels pour les fonctions est :

P1 := CréerFeuille(id, a)

P2 := CréerFeuille(nb, 4)

P3 := CréerNoeud('-', P1, P2)

P4 := CréerFeuille(id, c)

P5 := CréerNoeud('+', P3, P4)

L'arbre abstrait correspondant est donné par la figure 6.9. L'arbre abstrait physique est illustré dans la figure 6.10.

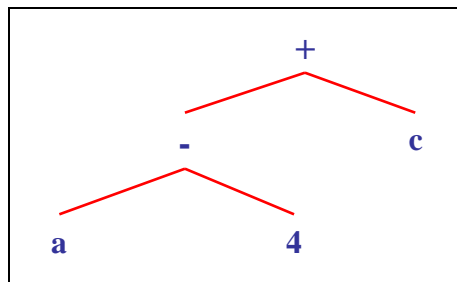


Figure 6.9. Arbre abstrait de l'expression  $a - 4 + c$

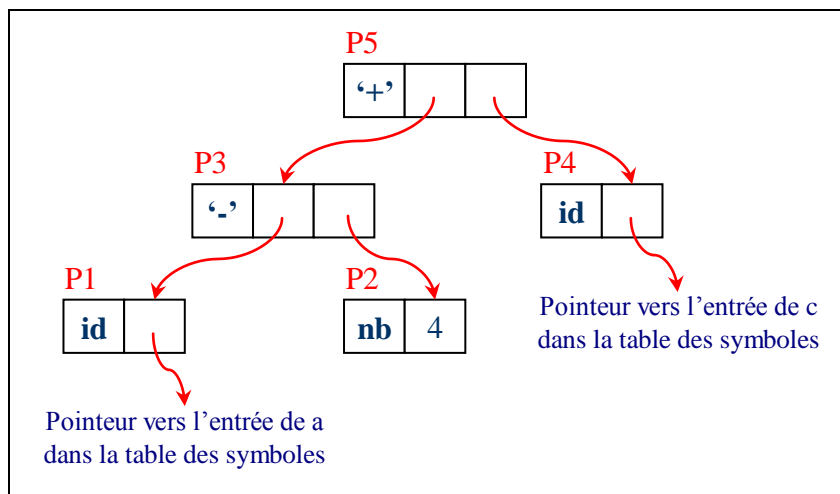


Figure 6.10. Représentation physique de l'arbre abstrait de l'expression  $a - 4 + c$

### Remarque

La construction est effectuée de manière ascendante.

### 6.3.3 Utilisation de la DDS pour la construction de l'arbre abstrait

La DDS suivante est une DDS s-attribuée pour la construction d'arbre abstrait d'expression contenant les symboles '+' et '-'.

Productions	Règles sémantiques
$E \rightarrow E1 + T$	$E.pnoeud := \text{CréerNoeud}('+', E1.pnoeud, T.pnoeud)$
$E \rightarrow E1 - T$	$E.pnoeud := \text{CréerNoeud}('-', E1.pnoeud, T.pnoeud)$
$E \rightarrow T$	$E.pnoeud := T.pnoeud$
$T \rightarrow (E)$	$T.pnoeud := E.pnoeud$
$T \rightarrow id$	$T.pnoeud := \text{CréerFeuille}(id, id.entree)$
$T \rightarrow nb$	$T.pnoeud := \text{CréerFeuille}(nb, nb.val)$

En utilisant la DDS précédente, la création de l'arbre abstrait pour l'expression «  $a - 4 + c$  » sera effectuée (de façon ascendante) de la manière suivante :

- Construction de l'arbre syntaxique concret
- Association des règles sémantiques aux règles de production correspondantes dans DDS

La figure 6.11 donne la représentation physique de l'arbre abstrait pour l'expression en question.

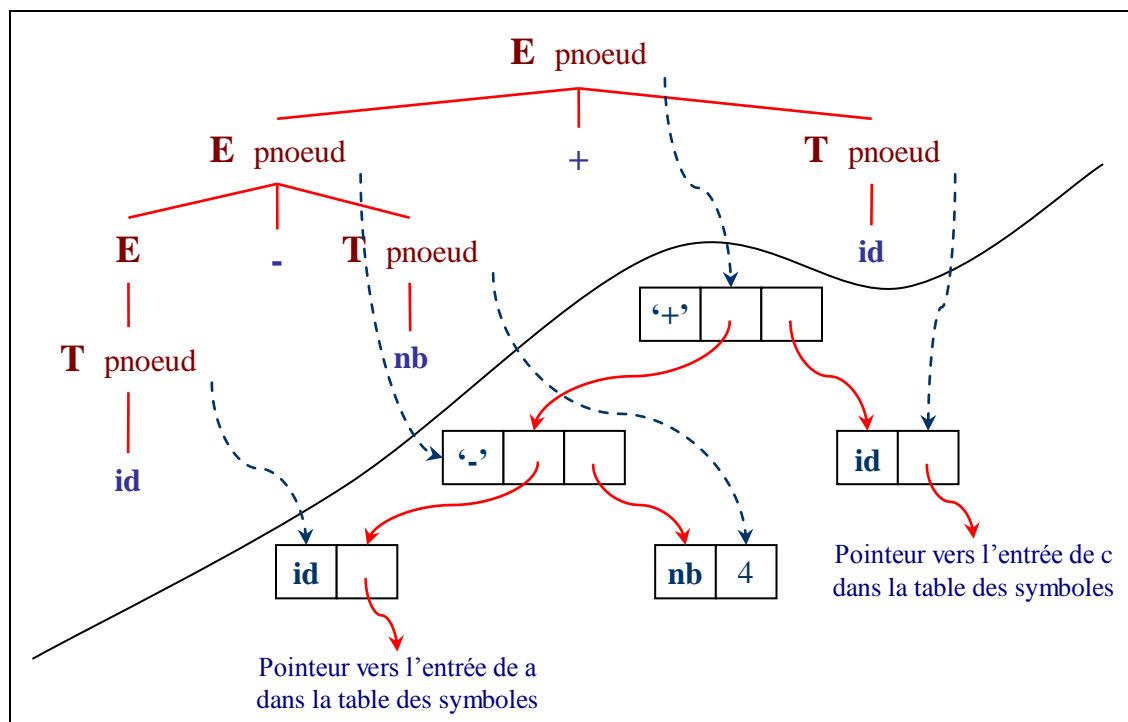


Figure 6.11. Construction de la représentation physique de l'arbre abstrait de l'expression  $a - 4 + c$

### 6.3.4 Graphe orienté acyclique pour les expressions

Un graphe orienté acyclique (DAG : Direct Acyclic Graph) correspondant à une expression identifie les sous-expressions communes qu'elle contient. Le DAG renferme un nœud pour chaque sous-expression. Les nœuds intérieurs représentent des opérateurs et leurs fils représentent les opérandes.

La différence entre DAG et arbre abstrait est qu'un nœud du DAG représentant une sous-expression commune a plus d'un père.

#### Exemple

La figure 6.12 donne l'arbre abstrait et le DAG pour l'expression:  $a + a * (b - c) + (b - c) * d$ .

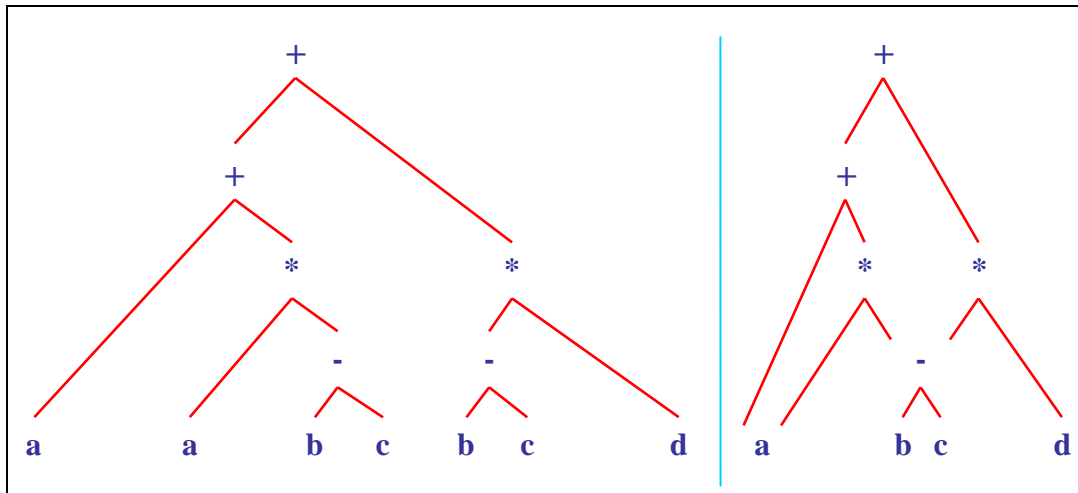


Figure 6.12. Arbre abstrait et DAG pour l'expression  $a + a * (b - c) + (b - c) * d$

Les DDS peuvent être utilisés pour construire des DAG de la même manière que pour les arbres abstraits. Les corps des fonctions créerNoeud et créerFeuille doivent être modifiés pour vérifier, avant la création, si le nœud ou la feuille considérée n'a pas déjà été créée. Dans le cas de l'existence, ces fonctions retournent un pointeur vers le nœud ou la feuille construite précédemment

## 6.4 Evaluation ascendante des définitions s-attribuées

Les attributs synthétisés peuvent être évalués par un analyseur syntaxique ascendant au fur et à mesure que le texte d'entrée est lu. L'analyseur syntaxique peut conserver dans sa pile les valeurs des attributs synthétisés associés aux symboles de la grammaire. A chaque fois qu'une réduction est effectuée, les valeurs des nouveaux attributs synthétisés sont calculées à partir des attributs des symboles en partie droite de la règle de production qui sont disponibles dans la pile.

On peut supposer que la pile est implémentée en utilisant deux tableaux « état » et « val ». Si état[i] contient le symbole « X » alors val[i] contient la valeur de l'attribut attaché au nœud de l'arbre syntaxique correspondant à ce « X ». A tout moment, le sommet de la pile est repéré par le pointeur sommet.

### Exemple

Productions	Règles sémantiques
$A \rightarrow XYZ$	$A.a := f(X.x, Y.y, Z.z)$

Avant la réduction de XYZ en A, Val[sommet] contient Z.z, Val[sommet-1] contient Y.y, Val[sommet-2] contient X.x

	<b>Etat</b>	<b>Val</b>
	X	X.x
	Y	Y.y
Sommet	Z	Z.z

Après la réduction de XYZ en A, le sommet est décrémenté de 2, état[sommet] contient A, val[sommet] contient A.a.

	<b>Etat</b>	<b>Val</b>
Sommet	A	A.a

## ***6.5. Evaluation en profondeur pour les attributs d'un arbre syntaxique***

---

Quand la traduction se fait pendant l'analyse syntaxique, l'ordre d'évaluation des attributs est liée à l'ordre de création des nœuds de l'arbre syntaxique par la méthode d'analyse. Pour beaucoup de méthodes de traduction (ascendantes ou descendantes), un ordre naturel d'évaluation est obtenu en appliquant à la racine d'un arbre syntaxique la procédure suivante :

**Procédure** VisiterEnProfondeur (n :noeud) ;

**Début**

**Pour** Tout fils m de n, de gauche à droite **faire**

**Début**

Evaluer les attributs hérités de m ;

VisiterEnProfondeur(m) ;

**Fin** ;

Evaluer les attributs synthétisés de m

**Fin** ;





## CHAPITRE 7 : TRADUCTION DIRIGEE PAR LA SYNTAXE ET GENERATION DE CODE INTERMEDIAIRE

### 7.1 Code à trois adresses

Les méthodes de traduction dirigées par la syntaxe sont utilisées pour produire les formes intermédiaires associées au langage de programmation. Parmi ces formes, on trouve les représentations graphiques telles que les arbres abstraits et les DAG ainsi que les représentations textuelles telles que le code à trois adresses qui est proche du langage d'assemblage. Dans ce code, chaque instruction contient en général trois adresses : 2 pour les opérandes et une pour le résultat.

Le code à trois adresses est une représentation intermédiaire bien adaptée à la complexité des problèmes des expressions arithmétiques et aux structures de contrôle imbriquées.

Le code à trois adresses correspond à une représentation linéaire des arbres abstraits et des DAG dans laquelle des noms explicites correspondent aux nœuds internes des graphes.

Un code à trois adresses est une séquence d'instructions de la forme  $x := y \text{ op } z$  où  $x, y$  et  $z$  sont des noms de constantes ou de variables temporaires produites par le compilateur.  $\text{op}$  étant un opérateur arithmétique ou logique.

#### Exemple

Soit à trouver la représentation sous forme d'arbre abstrait et de code à trois adresses de l'instruction d'affectation suivante :  $a := x + y * -z$

La figure 7.1 donne les représentations demandées :

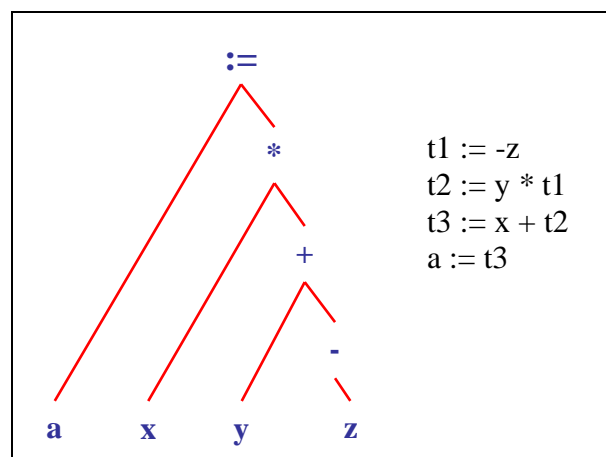


Figure 7.1. Représentations de l'affectation  $a := x + y * -z$

#### Remarque

Une instruction à trois adresses est une forme abstraite de code intermédiaire qui peut être implémentée dans un compilateur par des structures dont les champs contiennent l'opérateur, les opérandes et les résultats. Parmi ces structures, on trouve les quadruplets, les triplets et les triplets indirects.

## 7.2 Structure de quadruplet

Une structure de quadruplet est représentée par une table «TQ » contenant N lignes. Un quadruplet correspond à une ligne et consiste en quatre champs :

TQ	Opérateur	Argument 1	Argument 2	Résultat
i	*	y	t1	t2
N				

$t2 := y * t1$

### Remarque

Argument 1, Argument 2 et Résultat sont généralement des pointeurs sur les entrées (de la table des symboles) qui correspondent aux nœuds représentés par ces champs.

Dans un schéma de traduction utilisant les quadruplets, on peut faire appel à une procédure permettant de générer un quadruplet correspondant à  $Res := A1 \text{ op } A2$ .

**Procédure** Générer (oper :opérateur, A1, A2, Res : opérande) ;

#### Début

TQ[quadSuiv].Opérateur := oper ;

TQ[quadSuiv].Argument1 := A1;

TQ[quadSuiv].Argument2 := A2;

TQ[quadSuiv].Résultat := Res ;

quadSuiv := quadSuiv + 1 ;

**Fin** ;

### Remarque

TQ est la table de stockage des quadruplets.

quadSuiv : la prochaine ligne libre dans TQ.

Il existe aussi une fonction CréerTemp qui peut être utilisée dans les schémas de traduction, pour créer une variable temporaire dont le nom doit être différent de ceux utilisés par le programmeur.

**Fonction** CréerTemp (Var cpt : entier) : chaîne ;

#### Début

CréerTemp := '\$'+ConvEntChaîne(cpt) ;

cpt := cpt + 1 ;

**Fin** ;

cpt est un compteur de variables temporaires initialisé à zéro. ConvEntChaîne est une fonction qui convertit un entier en une chaîne de caractères.

Ainsi, CréerTemp est une fonction qui permet de générer séquentiellement des noms de variables temporaires \$0 ; \$1, \$2, ...

## 7.3 Traduction des expressions arithmétiques

Considérons la traduction des l'expression suivante en quadruplets :  $a * b + 3 * (b - c) + d$

Le code quadruplet est le suivant :

```
$0 := a * b
$1 := b - c
$2 := 3 * $1
$3 := $0 + $2
$4 := $3 + d
```

En analysant le code quadruplet obtenu, on constate que :

- Le nombre de quadruplets est égal au nombre d'opérateurs dans l'expression
- Pour chaque quadruplet, il faut créer une variable temporaire, donc, pour chaque règle de la grammaire contenant un opérateur, l'action sémantique associée doit contenir une instruction de création de variables temporaire et une instruction de génération de quadruplet.

### Schéma de traduction

Soit la grammaire des expressions arithmétiques :

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid id \mid nb$

Règle de production	Schéma de traduction
$E \rightarrow E_1 (+ - * /) E_2$	$x := \text{créerTemp}(\text{cpt})$ $E.\text{nom} := x$ Générer $((+ - * /), E_1.\text{nom}, E_2.\text{nom}, x)$
$E \rightarrow - E_1$	$x := \text{créerTemp}(\text{cpt})$ $E.\text{nom} := x$ Générer $(x := -E_1.\text{nom})$
$E \rightarrow (E_1)$	$E.\text{nom} := E_1.\text{nom}$
$E \rightarrow id$	$E.\text{nom} := id.\text{nom}$
$E \rightarrow nb$	$E.\text{nom} := nb.\text{nom}$

### Remarque

Pour pouvoir générer les quadruplets, il faut transmettre le nom de la variable résultat de l'expression  $E$  à l'unité syntaxique qui contient  $E$  par l'intermédiaire de la traduction du nom associé à  $E$ .

### Exemple

On se propose de construire l'arbre syntaxique de l'expression  $a * b + 3 * (b - c) + d$  avec l'évaluation de sa traduction en code quadruplet selon le schéma de traduction précédent.

On obtient le la séquence de code suivante :

```
$0 := a * b
$1 := b - c
$2 := 3 * $1
$3 := $0 + $2
$4 := $3 + d
```

La figure 7.2 donne l'arbre abstrait de l'expression :

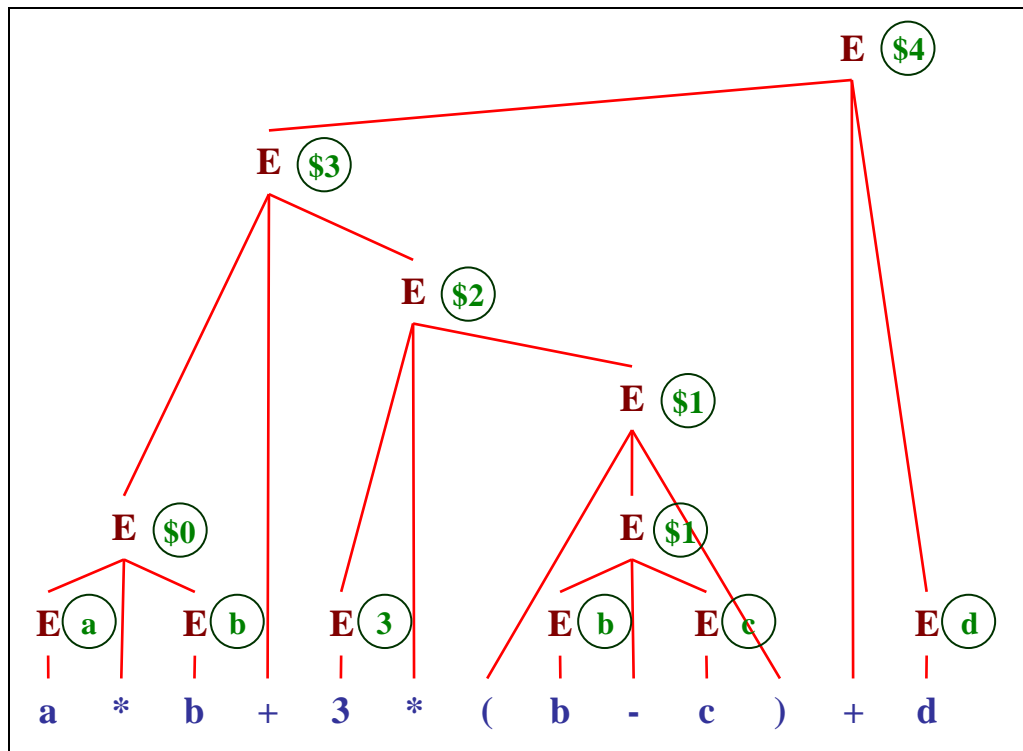


Figure 7.2. Arbre syntaxique de l'expression  $a * b + 3 * (b - c) + d$  avec l'évaluation de sa traduction en code quadruplet

## 7.4 Traduction des affectations de variables simples

Soit l'affectation  $a := (a + b) * c$ . Le code quadruplet qui lui correspond est :

```
$0 := a + b
$1 := $0 * c
a := $1
```

On remarque qu'il faut d'abord traduire l'expression  $(a + b) * c$  en code quadruplet puis générer le code quadruplet qui affecte le résultat à  $a$ .

id.nom étant le résultat de l'expression E.nom

A la production  $I \rightarrow id := E$  correspond le schéma de traduction : Générer (id.nom :=E.nom)

La figure 7.3 donne l'arbre de dérivation de l'affectation considérée.

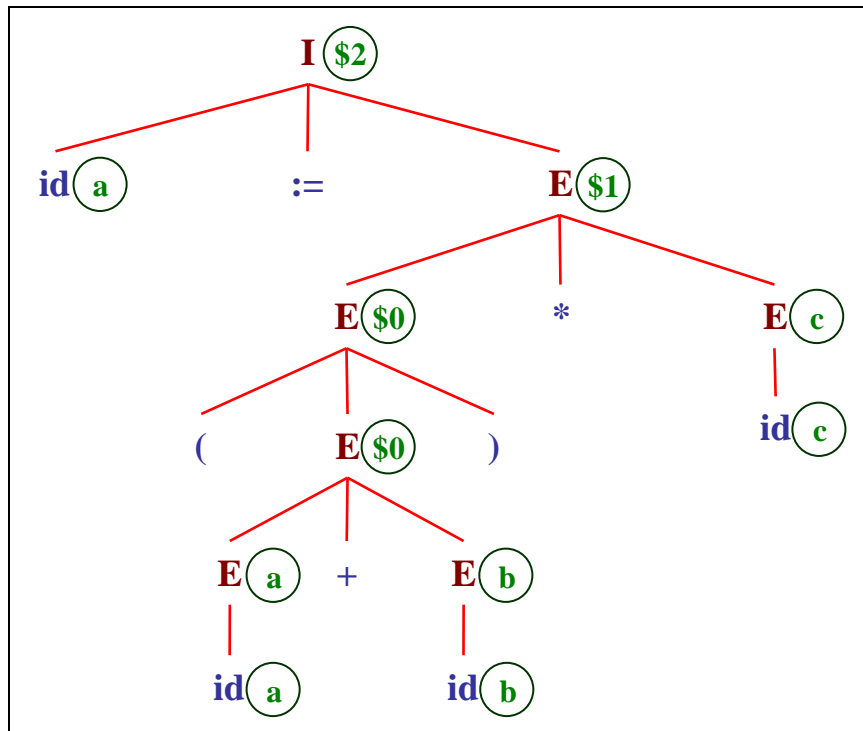


Figure 7.3. Arbre de dérivation de l'affectation  $a := (a + b) * c$  avec l'évaluation de sa traduction en code quadruplet

## 7.5 Traduction des expressions booléennes

Les expressions booléennes sont traduites de manière analogue aux expressions arithmétiques.

Traduisons en code quadruplet l'affectation suivante :  $c := a < b$

On associe la variable temporaire \$0 à l'expression  $a < b$  de telle sorte que si  $a$  est inférieur à  $b$ , \$0 sera égale à 1 et elle sera égale à 0 dans le cas contraire.

On obtient le code quadruplet suivant :

```

0 : if a < b goto 3
1 : $0 := 0
2 : goto 4
3 : $0 := 1
4 : c := $0
  
```

Soit la grammaire suivante, permettant de générer les expressions booléennes:

$E \rightarrow E \text{ and } E \mid E \text{ or } E \mid \text{not } E \mid (E) \mid \text{id} \mid \text{id OpRel id}$

$\text{OpRel} \rightarrow < \mid <= \mid > \mid >= \mid = \mid <>$

Le tableau suivant donne les schémas de traduction correspondant aux règles de production de la grammaire :

Règle de production	Schéma de traduction
$E \rightarrow E1 \text{ (and or) } E2$	$x := \text{créerTemp}(\text{cpt})$ $E.\text{nom} := x$ Générer ( $x := E1.\text{nom} \text{ (and or) } E2.\text{nom}$ )
$E \rightarrow \text{not } E1$	$x := \text{créerTemp}(\text{cpt})$ $E.\text{nom} := x$ Générer ( $x := \text{not } E1.\text{nom}$ )
$E \rightarrow (E1)$	$E.\text{nom} := E1.\text{nom}$
$E \rightarrow \text{id}$	$E.\text{nom} := \text{id}.\text{nom}$
$E \rightarrow \text{id1 OpRel id2}$	$x := \text{créerTemp}(\text{cpt})$ $E.\text{nom} := x$ Générer ( $\text{if id1.nom OpRel id2.nom goto quadSuiv} + 3$ ) Générer ( $x := 0$ ) Générer ( $\text{goto quadSuiv} + 2$ ) Générer ( $x := 1$ )

### Exemple

Soit à traduire en code quadruplet l'affectation suivante :

$a := b < c \text{ or } d > f \text{ and } g = h$

\$0	0 : if b < c goto 3 1 : \$0 := 0 2 : goto 4 3 : \$0 := 1
\$1	4 : if d > f goto 7 5 : \$1 := 0 6 : goto 8 7 : \$1 := 1
\$2	8 : if g = h goto 11 9 : \$2 := 0 10 : goto 12 11 : \$2 := 1
	12 : \$3 := \$1 and \$2 13 : \$4 := \$0 or \$3 14 : a := \$4

**SERIE DE TD N°:4 COMPILATION**  
**TRADUCTION DIRIGEE PAR LA SYNTAXE**

**Exercice 1**

Soit l'expression suivante :

$$a + a * (b - c) + (b - c) * d$$

1. Donner la représentation graphique de l'arbre abstrait et du DAG correspondant à cette expression
2. Donner la séquence d'appels permettant de construire ces deux représentations.

**Exercice 2**

Soit l'affectation suivante :

$$i := i + 10$$

1. Représenter graphiquement l'arbre abstrait et le DAG correspondant à cette affectation
2. Donner la séquence d'appels permettant de construire le DAG.
3. Donner la structure physique du DAG en supposant que les nœuds sont implémentés en mémoire de manière dynamique puis de manière statique.

**Exercice 3**

Donner le code quadruplet correspondant à l'affectation suivante :

$$x := (a+b*c) > (e*f/g) \text{ or not } h \text{ and not } (i \leq j)$$