Polycopié de Compilation

Prof. Abdelmajid DARGHAM

Octobre 2016

Préambule

Ce polycopié est destiné aux étudiants en informatique des Licences et Masters de l'Université Mohamed Premier. Il est écrit dans le but d'être utilisé comme support de cours du module Compilation de la Filière SMI $(Semestre\ 5)$. Il a comme objectif principal d'introduire les trois premières étapes de la phase d'analyse d'un compilateur, à savoir l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

Une version électronique de ce polycopié est également disponible dans la page du site web du cours à l'adresse : `http ://www.comp.sci.sitew.com`

Table des matières

Table des figures

Chapitre 1

Introduction à la compilation

Sommaire

1.1 Notion de compilation

Les langages de haut-niveau sont appropriés aux programmeurs humains et se caractérisent par une très grande différence du langage machine que l'ordinateur sait exécuter. Ainsi, des moyens pour combler le fossé entre les langages de haut-niveau et le langage machine sont nécessaires. C'est là où le compilateur entre en jeu.

1.1.1 Définition

Définition 1.1.1 *Un **compilateur** est un programme qui **traduit** un programme écrit dans un langage L vers un autre programme écrit dans un autre langage L'. Durant ce processus de traduction, le compilateur détectera également les erreurs évidentes du programmeur.*

La figure 1.1 illustre le schéma général d'un compilateur :

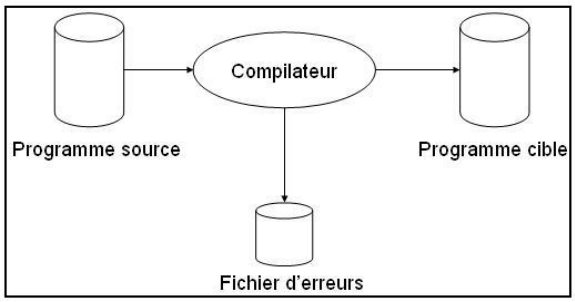

FIG. 1.1 – Schéma général d'un compilateur.

Le programme de départ s'appele **programme source** et le programme d'arrivé s'appele **programme cible**.

D'ordinaire, L est un langage de programmation de haut-niveau (C ou $Java$ par exemple) et L' est un langage de bas-niveau (par exemple un code machine adapté à un ordinateur particulier).

Exemple 1.1.1

- `gcc` *: compilateur de C vers de l'assembleur.*
- `javac` *: compilateur de $Java$ vers du byte code.*
- `latex2html` *: compilateur de $Latex$ vers du $Postcript$ ou du $HTML$.*

1.1.2 Avantages d'un compilateur

L'utilisation d'un compilateur permet de gagner plusieurs avantages :

▷ Il permet de programmer avec un niveau supérieur d'abstraction plutôt qu'en assembleur ou en langage machine.

▷ Il permet de faciliter la tâche de la programmation en permettant d'écrire des programmes lisibles, modulaires, maintenables et réutilisables.

▷ Il permet de s'eloigner de l'architecture de la machine et donc d'écrire des programmes portables.

▷ Il permet de détecter des erreurs et donc d'écrire des programmes plus fiables.

1.1.3 Les principales tâches d'un compilateur

▷ Lire et analyser le programme source.

▷ Détecter des erreurs (lexicales, syntaxiques et sémantiques).

▷ Écrire dans un langage intermédiaire la séquence d'instructions qui seront exécutées par la machine.

▷ Optimiser cette séquence d'instructions, notament la taille du code intermédiaire et sa vitesse d'éxecution.

▷ Traduire les instructions du langage intermédiaire dans le langage cible.

1.2 Les phases d'un compilateur

Traditionellement, un compilateur se découpe en trois grandes parties :

▷ Le **front-end** (ou **partie frontale**) qui se compose de trois **phases d'analyse** qui sont : l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

▷ Le **coeur** (ou **partie médiane**) qui travaille sur une **représentation intermédiaire** du programme, indépendante à la fois du langage source et du langage cible.

▷ Le **back-end** (ou **partie arrière**) qui contient trois **phases de synthèse** qui sont : la production du code intermédiaire, l'optimisation du code intermédiaire et la production du code cible.

La figure 1.2 montre le schéma global d'un compilateur travaillant avec cette division en trois parties. L'avantage de ce découpage est la réutilisabilité du coeur à la fois pour différents langages sources et différents langages cibles. Le rôle fondamental du coeur est la production d'une représentation intermédiaire entre le code source et le code cible. Cette représentation doit être plus facile à construire et à manipuler, mais aussi plus facile à traduire.

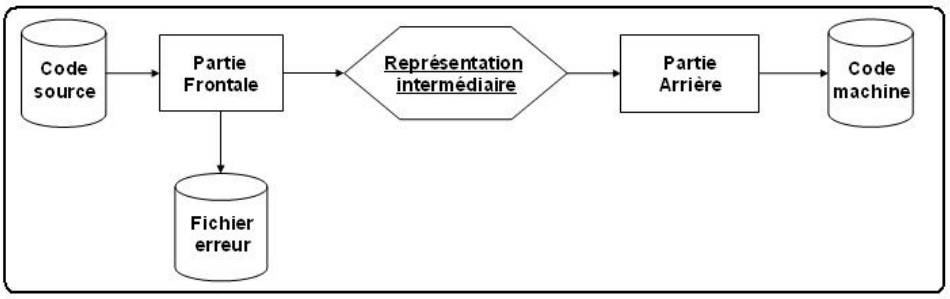

FIG. 1.2 – Division classique d'un compilateur.

1.2.1 Le front-end

Il comprend trois étapes d'analyse qui sont :

- L'**analyse lexicale** : elle prend en entrée le texte du programme source et fournit la liste de **lexèmes** (**jetons** ou **tokens**), avec pour chaque lexème son **unité lexicale** à laquelle il appartient (en plus d'autres informations). Le texte est lu caractère par caractère et divisé en **lexèmes**, qui correspond chacun à un mot valide dans le langage de programmation source. Un **mot-clé**, le **nom d'une variable**, une **constante numérique** sont des **lexèmes**.

- L'**analyse syntaxique** : elle prend la liste des lexèmes produite par l'analyse lexicale et regroupe ces derniers en une structure d'arbre (appelée l'**arbre de syntaxe**) qui reflète la structure grammaticale du programme source.

- L'**analyse sémantique** : elle prend en entrée l'arbre de syntaxe pour déterminer si le programme source viole certaines règles de cohérence. Par exemple, si une variable est utilisée mais non déclarée ou si elle est utilisée dans un contexte n'ayant pas un sens pour le type de la variable, comme l'addition d'un nombre avec une chaîne de caractères (valable en $Javascipt$, mais non valable en C).

1.2.2 Le back-end

Il comprend trois étapes de synthèse qui sont :

- La **production du code intermédiaire** : le programme source est traduit en une représentation intermédiaire simple qui est indépendante de la machine.

- L'**optimisation du code intermédiaire** : le code intermédiaire produit est

amélioré de façon à ce que le code machine résultant s'éxécute le plus rapidement possible.

- La **production du code cible** : le code intermédiaire optimisé est convertit en code cible qui est soit du code machine translatable ou du code en lanagage d'assemblage. Une tâche cruciale de cette étape est l'**allocation des registers** : les noms des variables symboliques utilisés dans le code intermédiaire seront traduits en nombres dont chacun correspond à un registre dans le code machine cible.

La figure 1.3 résume le schéma globale d'un compilateur avec ses étapes :

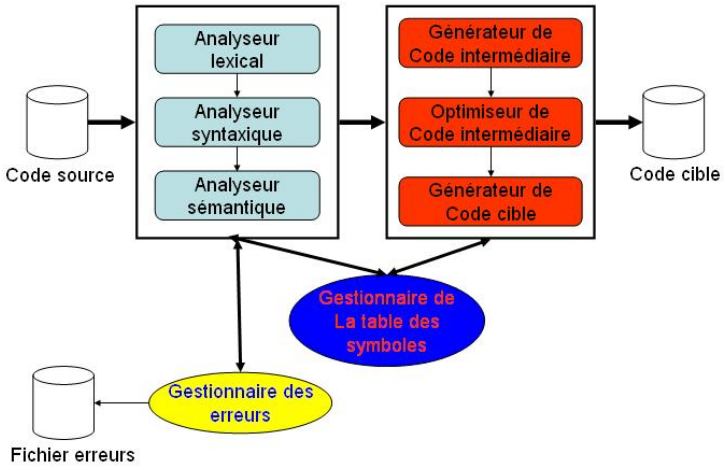

FIG. 1.3 – Schéma général d'un compilateur avec ses étapes.

1.2.3 Les différents modules

Le processus de compilation nécessite d'autres modules supplémentaires :

- Le **gestionnaire de la table des symboles** : une table de symbole est une table qui stocke les informations nécessaires sur les lexèmes. Le gestionnaire de la table des symboles est un programme qui gère cette table (construction de la table, insertions et recherches).

- Le **gestionnaire des erreurs** : il s'agit du module responsable de la détection et de la signalisation des erreurs rencontrées dans le programme source.

- ▷ Le **pré-processeur** : certains langages (comme C) ont besoin d'un module qui effectue un pré-traitement du texte source avant de déclencher le processus de compilation. Le plus souvent la tâche de ce module consiste à faire des recherche/remplacement de texte ou d'inclusion de certains fichiers.

- ▷ L'**assemblage** et l'**édition de lien** : le code en langage d'assemblage est traduit en représentation binaire et les addresses des variables et des fonctions sont fixées. Ces deux étapes sont typiquement effectués par des programmes fournis par la machine ou par le constructeur du système d'exploitation, et ne font pas partie du compilateur lui-même.

- ▷ Le **chargeur** : c'est un programme qui charge le code machine absolue dans la mémoire principale (la RAM) de l'ordinateur.

La figure 1.4 résume l'environnement d'un compilateur :

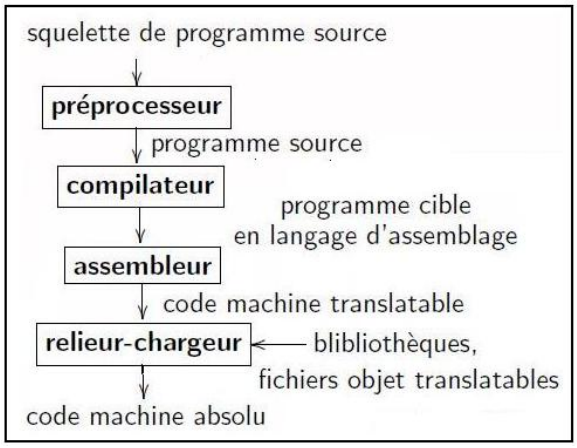

FIG. 1.4 – Schéma général de l'environnement d'un compilateur.

1.3 Les interpréteurs

L'**interprétation** constitue une autre solution pour implémenter un langage de programmation. Un **interpréteur**, au lieu de produire du code cible à partir de l'arbre de syntaxe, effectue lui-même les opérations spécifiées dans le code source. En fait, l'arbre de syntaxe sera traité directement pour évaluer les expressions et exécuter les instructions. Un interpréteur aura besoin de traiter la même pièce de l'arbre de syntaxe plusieurs fois (par exemple, le corps d'une boucle). Par

suite, l'interprétation est typiquement plus lente que l'exécution d'un programme compilé.

Exemple 1.3.1 *Basic, Visual Basic, JavaScript et Matlab sont des* ***langages interprétés***.

La compilation et l'interprétation peuvent être combinées pour implémenter un langage de programmation : le compilateur produira le code intermédiaire, lequel est ensuite interprété par un interpréteur (au lieu d'être compilé en code machine).

1.4 Caractéristiques d'un bon compilateur

Voici les caractéristiques d'un "**bon compilateur**" :

- ▷ Il doit générer un code objet optimisé en temps et en espace.
- ▷ Le temps de compilation est très rapide.
- ▷ Il doit déctecter toutes les erreurs lexicales et syntaxiques.
- ▷ Il doit détecter le maximum des erreurs sémantiques.
- ▷ Le nombre de passes de compilation est très réduit.
- ▷ Les messages d'erreurs sont clairs et précis.

Chapitre 2

Expressions régulières et automates finis

2.1 Langages formels

2.1.1 Alphabets

Définition 2.1.1 *Un **alphabet** est un ensemble A fini non vide $(A \neq \emptyset)$ de **symboles**.*

Par exemple, pour décire des nombres binaires, on utilise l'alphabet $A = \{0,\ 1\}$. L'**ASCII** et l'**EBCDIC** sont des exemples d'alphabet informatiques. Les symboles les plus couramment utilisés sont :

- Les lettres : a, b, ..., z, A, B, ..., Z, α, β, ..., ω
- Les chiffres : 0, 1, ..., 9
- Les symboles mathématiques : +, -, *, /, $\surd$, ...

2.1.2 Mots

Définition 2.1.2 *Un **mot** $(ou$ **chaîne**$)$ w sur un alphabet A, est une **séqeunce finie et ordonnée** de symboles de A. Si la suite des symboles de w est formée par les symboles respectifs : a_1, a_2, ..., a_n $(où\ a_i \in A$ dénote le i-ème symbole de $w)$, on écrit $w = a_1a_2...a_n$.*

Définition 2.1.3 *Si $w = a_1a_2...a_n$ est un mot, le nombre n représnte le nombre de symboles contenus dans le mot w et s'appelle la **longueur** du mot w. On le note aussi par $|w|$.*

Par exemple, 11010 est un mot sur l'alphabet $A = \{0,\ 1\}$ de longueur 5.

Définition 2.1.4 *Sur tout alphabet A, on peut définir un mot spécial : le **mot vide** noté ε, comme étant le mot de longueur 0 ($|\varepsilon| = 0$).*

2.1.3 Langages

Définition 2.1.5 *Un **langage** L sur un alphabet A est un ensemble (fini ou infini) de mots sur A.*

Par exemple, le langage des nombres binaires pairs et de longueur 3 est formé par les mots suivants : 000, 010, 100, 110.

Remarque 2.1.1 *Selon ces définitions, une unité lexicale est un langage sur un alphabet particulier. D'autre part, un alphabet A est considéré comme un langage sur A (les symboles sont des mots de longueur 1).*

Définition 2.1.6 *Soit A un alphabet. Le langage de tous les mots sur A est également un langage sur A. C'est le **langage universel** sur A, noté A^*.*

Remarque 2.1.2 *Il découle de cette définition :*

- ▷ *Une chaîne w est un mot sur un alphabet A, si et seulement si $w \in A^*$.*
- ▷ *Un ensemble L est un langage sur un alphabet A, si et seulement si $L \subseteq A^*$.*

2.1.4 Opérations sur les mots

Sur les mots, on peut définir plusieurs opérations :

- ▷ **Égalité** : deux mots u et v sur un même alphabet A sont égaux, si et seulement si $|u| = |v| = l$ et $u_i = v_i$, pour $i = 1, 2, ..., l$.
- ▷ **Concaténation** : soient u et v deux mots sur un même alphabet A. La **concaténation** de u et v qui s'écrit uv, est le mot formé en joignant les symboles de u et v. Par exemple, la concaténation de $u = 101$ et $v = 00$ est le mot $uv = 10100$.
- ▷ **Puissance** : soient w un mot sur un alphabet A et $n \geq 0$ un entier. La **puissance n-ème** du mot w qui se note w^n, est le mot obtenu par la concaténation

de n copies de w. Par exemple, $(101)^3 = 101101101$. Une **définition récursive** de la puissance n-ème d'un mot w est la suivante :

$$w^n = \begin{cases} \varepsilon & \text{si } n = 0 \\ w^{n-1}w & \text{si } n \geq 1 \end{cases} \tag{2.1}$$

▷ **Miroir** : le **miroir** d'un mot $w = a_1a_2...a_n$ est le mot $\widetilde{w} = a_n...a_2a_1$ obtenu en inversant les symboles de w. Par exemple, $\widetilde{aabaabc} = cbaabaa$. Un mot w est un **palindrôme** s'il est identique à son miroir, c'est-à-dire $\widetilde{w} = w$. Par exemple, $abba$ est un **palindrôme**.

Les opérations sur les mots vérifient les propriétés suivantes ($u,\ v,\ w$ étant des mots sur un alphabet A et $n,\ m$ des entiers) :

1. $|uv| = |u| + |v|$.
2. $u(vw) = (uv)w = uvw$: la concaténation est **associative**.
3. $w\varepsilon = \varepsilon w = w$: le mot vide ε est **neutre** pour la concaténation.
4. $|w^n| = n|w|$.
5. $w^{n+m} = w^n w^m$.
6. $(w^n)^m = w^{nm}$.
7. $|\widetilde{w}| = |w|$.
8. $\widetilde{uv} = \widetilde{v}\widetilde{u}$.
9. $\widetilde{\widetilde{w}} = w$: le miroir est une opération **involutive**.

Remarque 2.1.3 *La concaténation n'est pas commutative.*

2.1.5 Parties d'un mot

Soit w un mot sur un alphabet A.

▷ **Préfixe** : un mot u est un **préfixe** (ou **facteur gauche**) de w, si et seulement si, $w = uv$, pour un certain mot v. Autrement dit, le mot w commence par u.

▷ **Suffixe** : un mot v est un **suffixe** (ou **facteur droit**) de w, si et seulement s'il existe un mot u tel que $w = uv$. Autrement dit, le mot w se termine par v.

- ▷ **Facteur** : un mot u est un facteur de w, si et seulement s'il existe deux mots α et β tels que $w = \alpha u \beta$. Autrement dit, le mot u apparaît dans w.

- ▷ **Sous-mot** : un mot w' est un sous-mot de w, si et seulement si, w' peut être obtenu en supprimant un certain nombre (éventuellement 0) de symboles de w.

- ▷ **Préfixe,** (**Suffixe, Facteur et Sous-mot**) **propre** : un préfixe w' (resp. suffixe, facteur et sous-mot) d'un mot w est dit propre, si $w' \neq \varepsilon$ et $w' \neq w$.

Comme exemple, donnons les préfixes, les suffixes, les facteurs et les sous-mots du mot $w = 1011$ sur l'alphabet $A = \{0\,1\}$:

- Préfixes : ε, 1, 10, 101 et $1011 = w$.
- Suffixes : ε, 1, 11, 011 et 1011.
- Facteurs : ε, 0, 1, 01, 10, 11, 011, 101 et 1011.
- Sous-mot : ε, 0, 1, 01, 10, 11, 011, 101, 111 et 1011.

2.1.6 Opérations sur les langages

Les langages sont avant tout des ensembles, nous pouvons alors leur appliquer les opérations ensemblistes : réunion, intersection, complémentaire, différence et différence symétrique. Pour l'analyse lexicale, on s'intéresse principalement aux trois opérations suivantes : réunion, concaténation, fermeture et à leurs dérivées. Voici leurs définitions :

- ▷ **Réunion**. La **réunion** de deux langages L et M sur un même alphabet A notée $L \cup M$, est définie par :
$$L \cup M = \{w \in A^* \mid w \in L \text{ ou } w \in M\}$$
- ▷ **Concaténation**. La **concaténation** de deux langages L et M sur un même alphabet A notée LM, est définie par :
$$LM = \{uv \in A^* \mid u \in L \text{ et } v \in M\}$$
- ▷ **Puissance**. La puissance n-ème d'un langage L sur A, notée L^n, est le langage défini par récurrence par :
$$L^n = \begin{cases} \{\varepsilon\} & \text{si } n = 0 \\ L^{n-1}L & \text{si } n \geq 1 \end{cases} \tag{2.2}$$

D'une manière informelle, un mot $w \in L^n$ est obtenu par la concaténation de n mots de L.

▷ **Fermeture de Kleene**. La **fermeture de Kleene** (ou l'**étoile**) d'un langage L sur A, notée L^*, est le langage obtenu par la réunion de toutes les puissances de L :

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup ...$$

D'une manière informelle, un mot $w \in L^*$ est obtenu par la concaténation d'un nombre fini quelconque, éventuellemnt nul, de mots de L.

▷ **Fermeture positive**. La **fermeture de Kleene positive** d'un langage L sur A, notée L^+, est le langage obtenu par la réunion de toutes les puissances non nulles de L :

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup ...$$

D'une manière informelle, un mot de $w \in L^+$ est obtenu par la concaténation d'un nombre fini quelconque **non nul** de mots de L.

▷ **Miroir**. Le **miroir** d'un langage L sur A, noté $\widetilde{L}$, est le langage formé par tous les miroirs de ses mots :

$$\widetilde{L} = \{\widetilde{w} : w \in L\}$$

Voici quelques propriétés de ces opérations ($L,\ M,\ K$ étant des langages sur un alphabet A, et $n,\ m$ des entiers) :

1. $L(MK) = (LM)K = LMK$: la concaténation des langages est **associative**.
2. $L\{\varepsilon\} = \{\varepsilon\}L = L$: le langage réduit au mot vide $\{\varepsilon\}$ est **neutre** pour la concaténation des langages.
3. $L\emptyset = \emptyset L = \emptyset$.
4. $L(M \cup K) = (LM) \cup (LK)$ et $(L \cup M)K = (LK) \cup (MK)$: la concaténation est **distributive** par rapport à la réunion.
5. $L(M \cap K) \subseteq (LM) \cap (LK)$ et $(L \cap M)K \subseteq (LK) \cap (MK)$.
6. Si $K \subseteq M$, alors $LK \subseteq LM$ et $KL \subseteq ML$.
7. $L^{n+m} = L^n L^m$.
8. $(L^n)^m = L^{nm}$.

9. $(L^*)^* = L^*$.
10. $L^+ = LL^* = L^*L$.
11. $\widetilde{(LK)} = \widetilde{K}\widetilde{L}$.
12. $\widetilde{(L \cup K)} = \widetilde{L} \cup \widetilde{K}$.
13. $\widetilde{\widetilde{L}} = L$.
14. $\widetilde{(L^*)} = (\widetilde{L})^*$.

Remarque 2.1.4 *La concaténation des langages n'est pas commutative.*

Les opérations de réunion, de concaténation et de fermeture sont dites **opérations régulières**, car elles servent à écrire ce qu'on appelle des **expressions régulières** :

- La réunion exprime le **choix**.
- La concaténation exprime la **séquence**.
- La fermeture de Kleene exprime la **répétition**.
- La fermeture de Kleene positive exprime la **répétition non nulle**.

Par exemple, soit le langage L des mots sur l'alphabet $A = \{a,\ b,\ c\}$ qui commencent par a ou b, suivis de 0 ou plusieurs occurrences de cc et se termine par au moins un b. Il est clair que $L = L_1L_2L_3$, où $L_1 = \{a,\ b\}$, $L_2 = \{cc\}^*$ et $L_3 = \{b\}^+$. Finalement, $L = \{a,\ b\}(\{cc\}^*)\{b\}^+$, qui sera simplifié à $(a|b)(cc)^*b^+$.

2.2 Les expressions régulières

La forme des lexèmes d'une unité lexicale peut être décrite de manière informelle dans le manuel du langage. Par exemple, pour le langage C, un identificateur est une suite de lettres, de chiffres et de caractères de soulignement qui commence par une lettre ou un caractère de soulignement et qui ne soit pas un mot-clé du langage. Une telle description est satisfaisante pour le programmeur humain, mais certainement pas pour le compilateur.

Pour l'analyse lexicale, une spécification est traditionnellement écrite en utilisant une **expression régulière** : c'est une **notation algébrique compacte** pour décrire les unités lexicales. Cette notation est à la fois simple à comprender et à utiliser par les programmeurs humains, mais également manipulable d'une manière simple par un programme informatique.

2.2.1 Définition

La définition suivante explique sans ambigüité et d'une façon parallèle, ce qui est une **expression régulière** et le **langage** qu'elle **dénote** (représente). Pour noter une expression régulière, nous utilisons les symbles r, s, t, ...etc, et leurs langages $L(r)$, $L(s)$, $L(t)$, ...etc.

Définition 2.2.1 *Soit A un alphabet quelconque.*

- *$\emptyset$ est une expression régulière qui dénote le langage vide $\emptyset$.*
- *ε est une expression régulière qui dénote le langage $\{\varepsilon\}$ réduit au mot vide.* ***Attention****, les langages $\emptyset$ et $\{\varepsilon\}$ sont distincts ($\{\varepsilon\} \neq \emptyset$).*
- *Tout symbole $a \in A$ est une expression régulière qui dénote le langage $\{a\}$ (langage réduit au seul mot a, symbole de l'alphabet A).*
- *$r|s$ est une expression régulière qui dénote la réunion des deux langages dénotés respectivement par r et s, c'est-à-dire $L(r|s) = L(r) \cup L(s)$.*
- *rs est une expression régulière qui dénote la concaténation des deux languages $L(r)$ et $L(s)$, c'est-à-dire $L(rs) = L(r)L(s)$.*
- *r^* est une expression régulière qui dénote la fermeture de Kleene du language $L(r)$, c'est-à-dire $L(r^*) = (L(r))^*$.*

2.2.2 Règles de précédence

Lorsqu'on combine plusieurs symboles de construction dans une expression régulière, comme par exemple dans $a|ab^*$, il n'est pas clair comment les sous-expressions régulières sont regroupées. On peut alors utiliser les parenthèses $(...)$ pour fixer un regroupement de symboles, par exemple $(a|(ab))^*$. Pour simplifier l'écriture des expressions régulières, on adopte les règles de priorité suivantes :

1. l'opérateur de fermeture "$*$" a la plus haute priorité.
2. l'opérateur de concaténation a la deuxième plus haute priorité.
3. l'opérateur de réunion "|" a la plus faible priorité.

Les trois opérateurs s'associent de gauche à droite. Selon ces conventions, l'expression régulière $a|ab^*$ est équivalente à $(a|(a(b^*)))$.

La définition suggère de confondre une expression régulière s avec le langage $L(s)$ qu'elle dénote ($s \sim L(s)$). Comme exemple, le langage $\{a,\ b\}(\{cc\}^*)\{b\}^+$ sera confondu avec l'expression régulière qu'elle le dénote, à savoir $(a|b)(cc)^*b^+$.

2.2.3 Les notations abrégées

Il est utile d'utiliser certains raccourcis par convention :

- **intervalle de symboles** : nous écrivons par exemple $[0-9]$ au lieu de $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$. De même, l'ensemble des lettres minuscules est dénoté par $[a-z]$. Les entiers positifs sont décrits par l'expression régulière très courte, à savoir $[0-9][0-9]^*$.

- la **répétition non nulle** : nous écrivnos s^+ au lieu de $ss^* = s^*s$. Avec cette notation, on peut simplifier la description de l'ensemble des entiers positifs à $[0-9]^+$.

- l'**alternative** (0 ou une occurrence de) : nous écrivnos $s?$ au lieu de $(s \mid \varepsilon)$. Les opérateurs $+$ et $?$ ont la même priorité que l'opérateur $*$.

2.2.4 Quelques exemples

- **Mots-clés**. Un mot-clé (comme "**if**") est décrite par une expression régulière qui coincide avec ce mot-clé.

- **Identificateurs**. En C par exemple, un identificateur consiste en une séquence de lettres, chiffres et le caractère de soulignement et doit commencer par une lettre ou le symbole souligné. Cela peut être décrit par l'expression régulière $[a-zA-Z\_][a-zA-Z\_0-9]^*$.

- **Nombres entiers**. Une constante entière commence par un signe '$-$' (facultatif), et suivie d'une séquence non vide de chiffres, d'où l'expression régulière pour désigner des entiers en C : $-?[0-9]^+$.

2.2.5 Quelques propriétés algébriques

Définition 2.2.2 *Deux expressions régulières r et s, sur un même alphabet A, sont dites **équivalentes**, et l'on écrit $r = s$, si elles dénotent le même langage, c'est-à-dire si $L(r) = L(s)$.*

Par exemple, il est clair que $(a|b)^* = (b|a)^*$. Voici quelques propriétés algébriques des expressions régulières :

1. $(r|s)|t = r|(s|t) = r|s|t$: la réunion est associative.
2. $r|s = s|r$: la réunion est commutative.
3. $r|r = r$: la réunion est idempotent.
4. $r? = r|\varepsilon$: par définition de l'opérateur " ?".
5. $(rs)t = r(st) = rst$: la concaténation est associative.
6. $r\varepsilon = \varepsilon r = r$: l'élément ε est neutre pour la concaténation.
7. $r(s|t) = (rs)|(rt)$: la concaténation est distribituve à droite par rapport à la réunion.
8. $(r|s)t = (rt)|(st)$: la concaténation est distribituve à gauche par rapport à la réunion.
9. $(r^*)^* = r^*$: l'étoile est idempotent.
10. $r^*r^* = r^*$.
11. $rr^* = r^*r = r^+$: par définition de l'opérateur "+".

2.2.6 Définitions régulières

Pour des commodités de notation, on peut souhaiter donner des noms aux espressions régulières et définir des expressions régulières en utilisant ces noms comme s'ils étaient des symboles.

Définition 2.2.3 *Si A est un alphabet de base, une **définition régulière** est une suite de définition de la forme :*

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$...$$

$$d_n \rightarrow r_n$$

*où chaque d_i est un **nom** distinct et chaque r_i est une **expression régulière** sur $A \cup \{d_1,\ d_2, ...,\ d_n\}$.*

Pour distinguer un nom d_i d'une expression régulière r_i, on écrire le nom en gras et l'expression régulière en italique. Comme exemple, donnons une définition récursive d'un identificateur $Pascal$:

lettre $\rightarrow [A - Za - z]$
chiffre $\rightarrow [0 - 9]$
id $\rightarrow$ **lettre**(**lettre**|**chiffre**)*

2.3 Expressions régulières en Flex

Flex est un compilateur qui génère automatiquement des analyseurs lexicaux en C (voir chapitre 4). Il utilise des **expressions régulières** pour spécifier des **unités lexicales**. **Flex** offre un **ensemble plus étendu d'opérateurs** (autre que la concaténation, la réunion et l'étoile) pour décrire des expressions régulières. Voici la description de la majorité de ces opérateurs :

Expression Régulière	**Description**
x ou "x"	un seul caractère x
$r_1 r_2$	concaténation des expressions régulières r_1 et r_2
r^*	étoile de l'expression régulière r
r^+	étoile positive de l'expression régulière r
$r?$	équivalent à $(r\|\varepsilon)$
$[xyz]$	un caractère parmis ceux entre les crochets, c'est-à-dire x, y ou z
$[c-f]$	un caractère parmis ceux entre les crochets, mais entre c et f, c'est-à-dire c, d, e ou f ($-$, à l'intérieur de [ ], est l'opérateur d'intevalle)
.	n'importe quel caractère, sauf le caractère retour à la ligne '$\backslash n$'
$[\wedge\,]$	n'importe quel caractère, sauf ceux entre les crochets (opérateur de complémentation)
$r\{n,\ m\}$	de n à m fois l'expression régulière r, c'est-à-dire r^n, r^{n+1}, ..., r^m
$r\{n,\}$	un nombre de fois $\geq n$ l'expression régulière r, c'est-à-dire r^n, r^{n+1}, ..., etc
$r\{n\}$	exactement n fois l'expression régulière r, ou r^n
$r_1\|r_2$	réunion des expressions régulières r_1 et r_2
r_1/r_2	r_1, mais seulement si elle est suivie de r_2
$\wedge r$	r, mais sauf si elle est au début d'une ligne
$\$r$	r, mais sauf si elle est à la fin d'une ligne
$\backslash t$ ou "$\backslash t$"	un caractère tabulation
$\backslash n$ ou "$\backslash n$ "	un caractère retour à la ligne
{ }	opérateur de référencement d'une définition régulière
( )	opérateur de regroupement

- Les caractères d'échapements '\' et "..." sont utilisés pour désigner un opérateur de **Flex** (si ce dernier est un caractère dans le lexème).

- Les caractères '\', '$\wedge$' et '$' ne peuvent pas apparaître dans des ( ), ni dans des définitions régulières.

- À l'intérieur de [ ], l'opérateur '\' garde sa signification ainsi que l'opérateur '$-$' s'il n'est pas au début ou à la fin.

- Lorsque l'opérateur '$-$' est écrit au milieu d'un [ ], il signifie un intervalle.

- Le caractère de fin de fichier (`EOF` en C) est désigné en **Flex** par l'expression régulière $<< EOF >>$.

2.4 Langages réguliers

On ne peut pas décrire tous les langages par des expressions régulières. Par exemple, le langage $\{a^n b^n \mid n \geq 0\}$ ne peut pas être décrit par une expression régulière.

Définition 2.4.1 *Un langage L sur un alphabet A est dit **régulier** si et seulement s'il existe une **expression régulière** r qui le dénote, c'est-à-dire telle que $L = L(r)$).*

Exemple 2.4.1 *Les langages suivants sont réguliers :*

- *Le langage formé par les mots sur $\{0,\ 1\}$ qui se terminent pas 0 : il est dénoté par $(0|1)^*0$.*
- *Le langage formé par les mots sur $\{a, b\}$ ayant exactement une occurrence de b : il est dénoté par a^*ba^*.*
- *Le langage formé par les mots sur $\{a, b\}$ de longueur ≥ 2 : il est dénoté par $(a|b)(a|b)(a|b)^*$.*

Le résultat suivant découle immédiatement des propriétés des expressions régulièrs :

Théorème 2.4.1 *Soit A un alphabet quelconque.*

1. *$\emptyset$ et $\{\varepsilon\}$ sont réguliers.*
2. *$\forall a \in A$, le langage $\{a\}$ est régulier.*
3. *Tout le langage réduit à un seul mot est régulier.*
4. *Tout langage fini est régulier.*
5. *L'alphabet A est un langage régulier.*

2.4.1 Propriétés de clôture

Théorème 2.4.2 *Soit A un alphabet. L'ensemble des langages réguliers sur A est clôs par opérations régulières :*

1. ***Réunion** : si L et K sont réguliers, alors $L \cup K$ est régulier.*
2. ***Concaténation** : si L et K sont réguliers, alors LK est régulier.*
3. ***Etoile** : si L est régulier, alors L^* est régulier.*

Ce résultat découle aussi des propriétés sur les expressions régulières. Nous pouvons en déduire la propriété suivante :

Théorème 2.4.3 *Sur un alphabet A, le langage universel A^* est régulier.*

Remarque 2.4.1 *On montre également que l'ensemble des langages réguliers sur A est clos par les opérations suivantes :*

1. ***Intersection** : si L et K sont réguliers, alors $L \cap K$ est régulier.*
2. ***Différence** : si L et K sont réguliers, alors $L \setminus K$ est régulier.*
3. ***Miroir** : si L est régulier, alors $\widetilde{L}$ est régulier.*
4. ***Préfixe** : si L est régulier, alors $Pref(L)$ est régulier.*
5. ***Suffixe** : si L est régulier, alors $Suff(L)$ est régulier.*
6. ***Facteur** : si L est régulier, alors $Fact(L)$ est régulier.*

2.5 Les automates finis

Les **expressions régulières** constituent un **formalisme algébrique** pour spécifier les langages réguliers. Dans cette section, nous allons introduire un **formalisme graphique** qui nous permet de résoudre la question de la **reconnaissance** de ces langages.

Définition 2.5.1 *En général, un **reconnaisseur** pour un langage L sur un alphabet A est un* programme *qui prend en entrée une chaîne $w \in A^*$ et répond "oui" si $w \in L$ et "non" si $w \notin L$.*

Les **automates finis** constituent un outil formel très intéressant de la théorie des langages. Nous allons voir comment transformer une expression régulière en un reconnaisseur en construisant un automate fini. Il y a deux grandes catégories d'automates finis : les **AFD** (automates finis déterministes) et les **AFND** (automates finis non-déterministes).

Les **AFD** et les **AFND** sont capables de reconnaître précisément les **langages réguliers**. Alors que les **AFD** produisent des reconnaisseurs plus rapides que les **AFND**, un **AFD** est généralement beaucoup volumineux qu'un **AFND** équivalent.

2.6 Les automates finis déterministes

2.6.1 Définition

Définition 2.6.1 *Un **automate fini déterministe** (**AFD**), est un objet mathématique formé d'un* 5*-uplet* $M = (E,\ A, \delta, q_0,\ F)$ *où :*

- E *est un ensemble fini d'**états**.*
- A *est l'**alphabet** d'entrée.*
- δ *est la **fonction de transition** définie de* $E \times A$ *vers* E*.*
- $q_0 \in E$ *est l'**état initial**.*
- $F \subseteq E$ *est l'ensemble des **états finaux**.*

Un **AFD** dans le sens abstrait, est une "**machine**" qui possède un nombre fini d'**états** et un nombre fini de **transitions** entre ces états. Une transition entre deux états est étiquetté par un symbole de l'**alphabet d'entrée**. En théorie des langages, un **AFD** peut être utilisé pour décider si un mot w appartient à un langage donné L. Pour faire ceci, nous partons de l'état initial q_0 de l'autoamte. À chaque étape, on lit un symbole de w à partir de l'entrée, puis on suit une transition étiquettée par ce symbole et on change l'état courant. Il y a deux cas qui peuvent se présenter :

- ▷ Tous les symboles de l'entrée ont été lus. Nous testons alors si le dernier état est un état **final**. Si oui, le mot $w \in L$, sinon $w \notin L$.
- ▷ On se bloque sur un symbole particulier car il n'y a pas de transition avec ce symbole. Dans ce cas, $w \notin L$.

2.6.2 Représentation d'un AFD

Un **AFD** est représenté par un **graphe orienté étiquetté** :

- ▷ Les **états** sont les **sommets** du graphe et sont représentés par des **petits circles** portant des numéros (ou des noms) qui les identifient. Cependant, ces numéros ou noms n'ont aucune signification opérationnelle.

- ▷ Un **état final** est représenté par **deux cercles concentriques**.
- ▷ L'**état initial** q_0 est marqué par une **flèche qui pointe sur son cercle**.
- ▷ Les **transitions** représentent les **arcs** du graphe. Une transition est dénotée par une flèche reliant deux états. Cette flèche porte une étiquette qui est un symbole de l'alphabet.

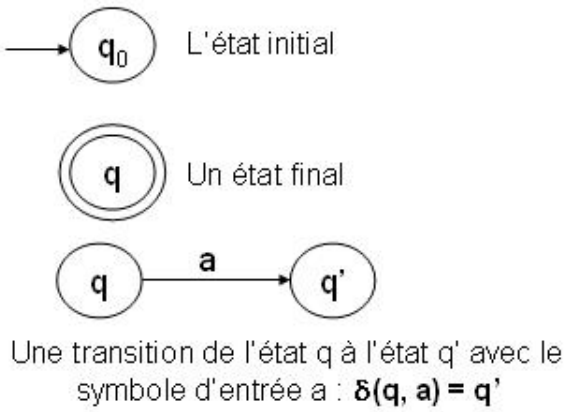

FIG. 2.1 – Représentation graphique d'un **AFD**.

Prenons l'exemple de l'expression régulière a^*ba^*. Elle dénote le langage sur l'alphabet $\{a,\ b\}$ des mots ayant exactement une seule occurrence de b. Ce langage est reconnu par l'**AFD** suivant :

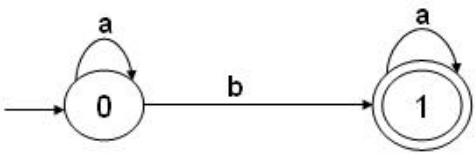

FIG. 2.2 – Un **AFD** reconnaissant le langage a^*ba^*.

L'**AFD** de l'exemple est défini formellement par les éléments suivants :

- $E = \{0,\ 1\}$.
- $A = \{a,\ b\}$.

- Trois transitions : $\delta(0,\ a) = 0$ (c'est une boucle), $\delta(0,\ b) = 1$ et $\delta(q_1,\ a) = 1$ (une autre boucle).

- L'état initial $q_0 = 0$.

- $F = \{1\}$.

La fonction de transition δ d'un **AFD** peut être représentée à l'aide d'un tableau bidimensionel appelé **matrice de transition**. Les lignes de cette matrice sont indexées par les états et les colonnes par les symboles de l'entrée. Une case $\delta(q,\ a)$, où $q \in E$ et $a \in A$ représente l'état d'arrivée de la transition avec le symbole a qui part de l'état q. Si cette transition n'est pas définie, cette case est laissée vide. La matrice de transition de l'**AFD** de l'exemple est la suivante :

δ	a	b
0	0	1
1	1	

Le mot baa appartient au langage a^*ba^*, en effet :

- On part de l'état initial $q_0 = 0$. On lit le premier symbole de l'entrée, à savoir "b". Il y a une transition de l'état 0 avec le symbole "b", on la suit et on passe à l'état 1.

- On lit le second symbole de l'entrée, à savoir "a". Il y a une transition de l'état 1 avec le symbole "a", on la suit et on reset à l'état 1.

- On lit le troisième symbole de l'entrée, à savoir "a". Il y a une transition de l'état 1 avec le symbole "a", on la suit et on reset à l'état 1. On a terminé la lecture de tous les symboles de l'entrée, on teste le dernier état. Comme $1 \in F$, alors le mot $baa \in a^*ba^*$.

Le mot aa n'appartient pas au langage a^*ba^*, en effet :

- On part de l'état initial $q_0 = 0$. On lit le premier symbole de l'entrée, à savoir "a". Il y a une transition de l'état 0 avec le symbole "a", on la suit et on reste à l'état 0.

- On lit le second symbole de l'entrée, à savoir "a". Il y a une transition de l'état 0 avec le symbole "a", on la suit et on reset à l'état 0. On a terminé

la lecture de tous les symboles de l'entrée, on teste le dernier état. Comme $0 \notin F$, alors le mot $aa \notin a^*ba^*$.

Le mot bba n'appartient pas au langage a^*ba^*, en effet :

- On part de l'état initial $q_0 = 0$. On lit le premier symbole de l'entrée, à savoir "b". Il y a une transition de l'état 0 avec le symbole "b", on la suit et on passe à l'état 1.

- On lit le deuxième symbole de l'entrée, à savoir "b". Il n'y a pas de transition de l'état 1 avec le symbole "b", on est bloqué sur le troisième symbole et le mot $bba \notin a^*ba^*$.

Remarque 2.6.1
*La fonction de transition d'un **AFD** est **partielle**, c'est-à-dire que $\delta(q,\ a)$ n'est pas toujours défini pour tout état q et tout symbole a. Cependant, il est possible de la rendre totale par le procédé suivant :*

1. *On ajoute un nouveau état β à E (un état **artificiel**).*
2. *On met $\delta(q,\ a) = \beta$ pour tout état q et tout symbole a pour lesquels $\delta(q,\ a)$ est non défini.*
3. *On ajoute les transitions $\delta(\beta,\ a) = \beta$ pour tout symbole a.*

*L'**AFD** obtenu par cette transformation est équivalent à l'**AFD** initial, en ce sens qu'ils reconnaissent le même langage. Un tel **AFD** est dit **complet**.*

À titre d'exemple, voici l'**AFD** complet équivalent à l'**AFD** de la figure 2.2 :

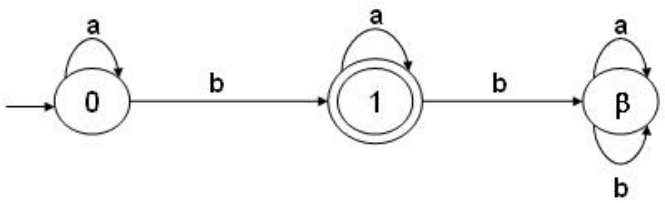

FIG. 2.3 – L'**AFD complet** équivalent à l'**AFD** de la figure 2.2.

2.6.3 Reconnaissance d'un mot par un AFD

Définition 2.6.2 *Par définition, un mot w est **reconnu** par un **AFD** $M = (E,\ A,\ \delta,\ q_0,\ F)$, s'il existe un **chemin** C étiquetté par w et allant de l'état initial q_0 vers un état final $q \in F$ de M. Formellement, un mot $w = a_1...a_n$ est reconnu par l'**AFD***

M, s'il existe n transitions de la forme suivante : $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2$, ..., $\delta(q_{n-1}, a_n) = q_n$ et $q_n \in F$.

Exemple 2.6.1 *Les mots b, ab et $aabaaa$ sont reconus par l'**AFD** de la figure 2.3.*

Une autre définition équivalente de la reconnaissance d'un mot par un **AFD** est basée sur la **fonction de la transition itérée** notée δ^*.

Définition 2.6.3 *La **fonction de transition itérée** d'un **AFD** complet est la fonction totale définie de $E \times A^*$ vers E par :*

$$\delta^*(q,\ \varepsilon) = q$$

$$\delta^*(q,\ aw) = \delta^*(\delta(q,\ a),\ w)$$

où $q \in E$ est un état, $a \in A$ est un symbole et $w \in A^*$ un mot.

Exemple 2.6.2 *Dans l'**AFD** de la figure 2.3, on a : $\delta^*(0,\ baa) = 1$ et $\delta^*(0,\ aa) = 0$, mais $\delta^*(0,\ abba) = \beta$.*

Définition 2.6.4 *Un mot w est **reconnu** par un **AFD** $M = (E,\ A,\ \delta, q_0, F)$, si et seulement si, $\delta^*(q_0, w) \in F$.*

Définition 2.6.5 *Le langage reconnu par un **AFD** M, noté $L(M) = (E,\ A,\ \delta, q_0, F)$, est le langage formé par tous les mots reconnus par M. Formellement :*

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

Exemple 2.6.3 *Le langage reconnu par l'**AFD** de la figure 2.3 est a^*ba^*.*

La preuve est évidente si l'on utilise la formule donnée par la proposition suivante :

Proposition 2.6.1 *Dans un **AFD** complet $M = (E,\ A,\ \delta,\ q_0,\ F)$, on a :*

$$\delta^*(q,\ uv) = \delta^*(\delta^*(q,\ u),\ v)$$

pour tout mots u et v de A^*, et tout état $q \in E$.

Remarque 2.6.2 *Dans un **AFD**, le chemin de reconnaissance d'un mot est unique.*

Algorithm 1 Reconnaissance d'un mot w par un AFD M

```
1: etat := q0;
2: n := |w|;
3: for (i = 0; i < n; i++) do
4:     etat := δ(etat, w[i]);
5:     if (etat non défini) then
6:         return 0;
7:     end if
8: end for
9: if (etat ∈ F) then
10:    return 1;
11: else
12:    return 0;
13: end if
```

FIG. 2.4 – Algorithme de reconnaissance d'un mot par un **AFD**.

2.6.4 Algorithme de reconnaissance d'un mot par un AFD

L'algorithme de reconnaissance d'un mot par un **AFD** est plus simple et plus efficace (son temps d'exécution est linéaire en la longueur du mot). Nous présentons ici le pseudo-code de cet algorithme. Ce dernier prends un mot $w \in A^*$ et un **AFD** $M = (E,\ A,\ \delta,\ q_0,\ F)$ et retourne 1 si $w \in L(M)$ et 0 sinon.
Notre objectif est de transformer une **expression régulière** en un **reconnaisseur**. Il suffit alors de trouver un **automate fini** à partir d'une telle expresssion régulière. Malheureusement, l'obtention d'un **AFD** à partir d'une expession régulière n'est pas toujours garantie et n'est pas automatique. L'utilisation des **AFND** a été choisie car il est plus simple (et même automatique) de construire un **AFND** à partir d'une expression régulière (**ER**) que de construire un **AFD**. Les **AFND** constituent alors un bon intermédiaire entre les **ER** et les **AFD**. Ce qui est rassurant, c'est que le passage d'un **AFND** à un **AFD** est également automatique et plus simple en pratique (même s'il est théoriquement exponnentiel).

2.7 Les automates finis non-déterministes : AFND

Les **automates finis non-déterministes** (**AFND**) constituent un outil formel efficace pour transformer des expressions régulières en programmes efficaces de reconnaissance de langages. Par leur nature **non-déterministe**, ils sont plus compliqués que les **AFD**.

2.7.1 Définition d'un AFND

La nature **non-déterministre** d'un **AFND** provient des points suivants :

- ▷ Il peut posséder plusieurs états initiaux.
- ▷ Il peut y avoir plusieurs tarnsitions à partir d'un même état avec le même symbole de l'entrée.
- ▷ Il peut y avoir des **tarnsitions spontanées** (c'est-à-dire sans lire aucun symbole d'entrée) d'un état vers un autre.

Définition 2.7.1 *Une ε-**transition** (transition **spontanée**, ou transition **immédiate**) est le passage d'un état q vers un autre état q' sans lire aucun symbole. Dans ce cas, on dit que l'automate a lit le mot vide ε.*

Par sa nature, le chemin de reconnaissance d'un mot par un **AFND** n'est pas unique. Voici la définition formelle d'un **AFND** :

Définition 2.7.2 *Un **AFND** est un objet mathématique formé d'un 5-uplet $M = (E,\ A,\ \delta,\ Q_0,\ F)$ où :*

- ▷ *E est un ensemble fini d'**états**.*
- ▷ *A est l'**alphabet d'entrée**.*
- ▷ *$Q_0 \subseteq E$ est l'ensemble des **états initiaux**.*
- ▷ *δ est la **fonction de transition** définie de $E \times (A \cup \{\varepsilon\})$ vers $\wp(E)$ ($\wp(E)$ désigne l'ensemble des parties de E).*
- ▷ *$F \subseteq E$ est l'ensemble des **états finaux**.*

La fonction de transition δ associe à un couple formé d'un état de départ $q \in E$ et un symbole d'entrée $a \in A$ (ou ε), un sous-ensemble d'états d'arrivé $\delta(q,\ a) \subseteq E$. Un **AFND** se représente de la même façon qu'un **AFD**, sauf que les ε-**transitions** sont étiquettées par le mot vide ε. La figure suivante montre un exemple d'un **AFND** :
Formellement, cet **AFND** est défini par les éléments suivants :

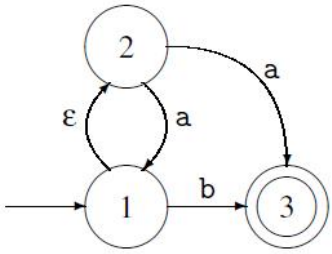

FIG. 2.5 – Exemple d'un AFND.

- $E = \{1,\ 2,\ 3\}$.
- $A = \{a,\ b\}$.
- $Q_0 = \{1\}$.
- La **matrice de transition** :

δ	a	b	ε
1		$\{3\}$	$\{2\}$
2	$\{1,\ 3\}$		
3			

- $F = \{3\}$.

2.7.2 Reconnaissance d'un mot par un AFND

Définition 2.7.3 *Soit $M = (E,\ A,\ \delta,\ Q_0,\ F)$ un **AFND** et $w \in A^*$ un mot sur A. On dit que M **reconnaît** (ou **accèpte**) le mot w, s'il existe un chemin dans M allant d'un état initial $q_0 \in Q_0$ vers l'un des états finaux de M qui correspond à la séquence des symboles du mot w. Plus formellement, il existe une séquence d'états $e_1,\ e_2,\ ...,\ e_{n+1}$ et une séquence de symboles $a_1,\ a_2,\ ...,\ a_n$ telles que :*

1. $e_1 = q_0 \in Q_0$.
2. $e_{j+1} \in \delta(e_i,\ a_i)$, *pour* $i = 1, 2, ..., n$.
3. $a_1 a_2 ... a_n = w$.
4. $e_{n+1} \in F$.

Par exemple, l'**AFND** de la figure 2.5 reconnaît le mot aab :

- On part de l'état initial 1. On lit le symbole ε et on passe à l'état 2.
- On lit le symbole a et on passe à l'état 1.
- On lit le symbole ε et on passe à l'état 2.
- On lit le symbole a et on passe à l'état 1.
- On lit le symbole b et on passo à l'état 3.
- On a lit tous les symboles du mot et le dernier état est $3 \in F$.

Par contre, le même automate n'accèpte pas le mot bb.

Définition 2.7.4 *Soit $M = (E,\ A,\ \delta,\ Q_0,\ F)$ un **AFND**. Le **langage reconnu** (ou accepté) par M, noté $L(M)$, est le langage formé par tous les mots reconnus par M.*

Exemple 2.7.1 *Le langage reconnu par l'automate de la figure 2.5 est $a^*(a|b)$.*

Un programme qui décide si un mot est accépté par un **AFND** donné, doit vérifier tous les chemins possibles pour tester l'accéptation du mot. Cela nécessite alors d'effectuer des retours en arrière (**backtracking**) jusqu'à trouver un chemin favorable. Comme un algorithme qui fait des retours en arrière est le plus souvent non efficace, les **AFND** sont des mauvais reconnaisseurs.

2.8 Conversion d'une ER en un AFND

L'**algorithme de Thompson** donne un moyen pour convertir automatiquement une expression régulière r dénotant un langage L en un **AFND** ayant certaines propriétés et qui accèpte le même langage L.

2.8.1 AFND normalisé

L'**algorithme de Thompson** construit en fait un **AFND normalisé**. Ce dernier est un cas spécial d'un **AFND** :

1. Il possède un seul état initial q_0.
2. Il possède un seul état final $q_f \neq q_0$.
3. Aucune transition ne pointe sur l'état initial.

4. Aucune transition ne sorte de l'état final.
5. Tout état, est soit l'origine d'exactement une transition étiquettée par un symbole de l'alphabet, soit l'origine d'au plus deux transitions, étiquettées par le mot vide ε.

2.8.2 L'algorithme de Thompson

L'algorithme est **récursif** : on décompose l'expression régulière en sous-expressions élémentaires avec les opérations régulières, puis on applique récursivement le procédé suivant :

1. Pour l'expression régulière $r = \emptyset$, construire l'**AFND** :

FIG. 2.6 – Automate reconnaîssant $\emptyset$.

2. Pour l'expression régulière $r = \varepsilon$, construire l'**AFND** :

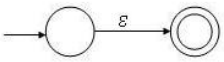

FIG. 2.7 – Automate reconnaîssant ε.

3. Pour l'expression régulière $r = a \in A$, construire l'**AFND** :

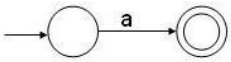

FIG. 2.8 – Automate reconnaîssant un symbole $a \in A$.

4. Supposons que $M(r)$ et $M(s)$ soient les **AFND** obtenus par l'**algorithme de Thompson** pour les expressions régulières r et s, respectivement.
   (a) Pour l'expression régulière $(r|s)$, construire l'**AFND** :
   (b) Pour l'expression régulière (rs), construire l'**AFND** :
   (c) Pour l'expression régulière $(r)^*$, construire l'**AFND** :

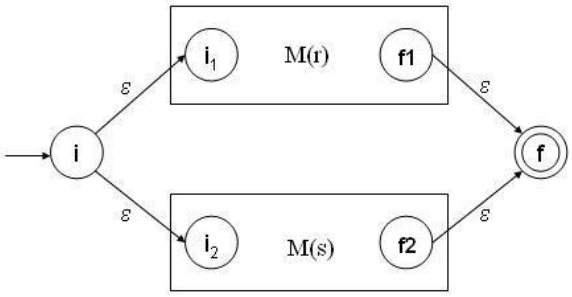

FIG. 2.9 – Automate reconnaîssant la réunion $(r|s)$.

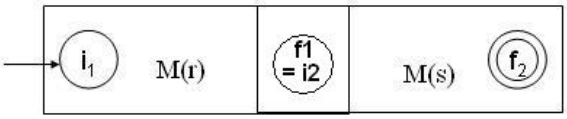

FIG. 2.10 – Automate reconnaîssant la concaténation (rs).

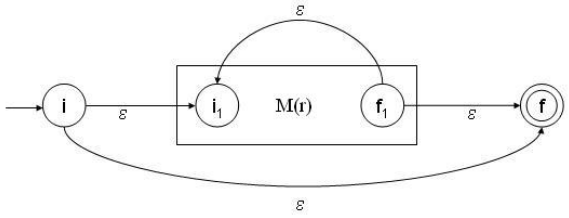

FIG. 2.11 – Automate reconnaîssant l'étoile $(r)^*$.

Théoriquement, on montre que le nombre des états de l'**AFND** obtenu ne dépasse pas le double de la taille de l'expression régulière. Cette taille, notée $\| \ldots \|$, est calculée récursivement de la manière suivante :

1. $\| \emptyset \| = \| \varepsilon \| = 0$.
2. $\| a \| = 1, \forall a \in A$.
3. $\| (r|s) \| = \| r \| + \| s \| + 1$.
4. $\| (rs) \| = \| r \| + \| s \| + 1$.
5. $\| (r)^* \| = \| r \| + 1$.

Exemple 2.8.1 *La figure suivante montre l'**AFND** obtenu par l'application de l'**algorithme de Thompson** à l'expression régulière a^*ba^*.*

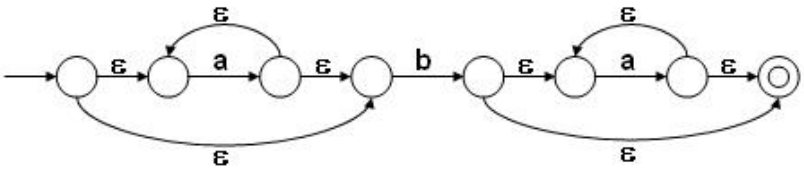

FIG. 2.12 – Automate reconnaissant l'expresion a^*ba^*.

2.9 Conversion d'un AFND en un AFD

La transformation est faite en utilisant une **technique de regroupement**. L'idée générale est que chaque état dans l'**AFD** correspond à un ensemble d'états dans l'**AFND**. L'**AFD** utilise un état pour garder trace de tous les états possibles que l'**AFND** peut atteindre après avoir lu chaque symbole de l'entrée. Ceci revient à dire que, après avoir lu le mot $a_1a_2...a_n$, l'**AFD** est dans un état qui représente le sous-ensemble T des états de l'**AFND accessibles** à partir de l'état de départ de l'**AFND** en suivant le chemin $a_1a_2...a_n$. Le nombre d'états de l'**AFD** peut être exponentiel par rapport au nombre d'états de l'**AFND**, mais en pratique, ce cas le plus défavorable apparaît rerement.

2.9.1 ε-clôture

Les ε-transitions compliquent un peu cette construction : chaque fois qu'on a un état de l'**AFND**, on peut toujours choisir de suivre une ε-transition. Ainsi, étant donné un symbole, un état d'arrivé peut être trouvé, soit en suivant une transition étiquettée avec ce symbole, soit en faisant un certain nombre d'ε-transitions puis une transition avec le symbole. On peut résoudre ce cas de figure dans la construction, en étendant en premier lieu l'ensemble des états de l'**AFND** avec ceux accessibles à partir de ces états en utilisant uniquement des ε-transitions. Ainsi, pour chaque symbole de l'entrée possible, on suit les transitions avec ce symbole pour former un nouveau sous-ensemble d'états de l'**AFND**.

On définit l'ε-**clôture** (ou l'ε-**fermeture**) d'un sous-ensemble d'états T, noté $\widehat{\varepsilon}(T)$, comme étant la réunion de T et de l'ensemble de tous les états accessibles depuis l'un des états de T en utilisant uniquement un certain nombre d'ε-**transitions**. Formellement :

Définition 2.9.1 *Soit $T \subseteq E$, où E est l'ensemble des états d'un **AFND** donné. $\widehat{\varepsilon}(T) = T \cup \{q' \in E \mid$ il existe un chemin étiqueté par des ε allant d'un état $q \in T$ vers l'état $q'\}$.*

Exemple 2.9.1 *Pour l'un **AFND** de la figure* 2.5*, on a :*

$$\widehat{\varepsilon}(\{1,\ 3\}) = \{1,\ 2,\ 3\}$$

2.9.2 Algorithme de calcul de la ε-clôture

Le calcul de la ε-clôture d'un sous-ensemble détats T est un processus typique de recherche dans un graphe d'un ensmeble donné de nœud. Dans ce cas, les états de T forment l'ensemble donné de nœuds et le graphe est constitué uniquement des ε-transitions de l'**AFND**. Un algorithme simple pour calculer la ε-clôture de T utilise une pile pour conserver les états dont les transitions sur ε n'ont pas encore été examinés. Voici le pseudo-code de cet algorithme :

Algorithm 2 Calcul de la ε-clôture d'un ensemble d'états T

```
1: Empiler tous les états de T dans la pile P;
2: Initialiser ε̂(T) à T : ε̂(T) ← T;
3: while (la pile P est non vide) do
4:     t ← Dépiler(P);
5:     for each u avec une ε-transition de t à u do
6:         if u ∉ ε̂(T) then
7:             Ajouter u à ε̂(T);
8:             Empiler(u, P);
9:         end if
10:    end for
11: end while
```

FIG. 2.13 – Algorithme de calcul de la $\widehat{\varepsilon}$-clôture d'un ensemble détats T.

2.9.3 Algorithme de déterminisation d'un AFND

L'algorithme de **déterminisation** prend en entrée un **AFND** $M = (E,\ A,\ \delta,\ Q_0,\ F)$ et fournit en sortie un **AFD** $M' = (E',\ A',\ \delta',\ q'_0,\ F')$ équivalent à M (c'est-à-dire tel que $L(M) = L(M')$). Le pseudo-code de cet algorithme est présenté dans la figure suivante :

Exemple 2.9.2 *Appliquons cet algorithme pour convertir l'**AFND** de la figure* 2.5 *en un **AFD** équivalent :*

1. $A' = A = \{a,\ b\}$.

Algorithm 3 Dterminisation d'un **AFND**

1: $A' \leftarrow A$;
2: $q'_0 \leftarrow \widehat{\varepsilon}(Q_0)$;
3: Ajouter q'_0 à E';
4: Marqué$(q'_0) \leftarrow false$;
5: **while** (il existe un état non marqué T dans E') **do**
6: Marqué$(T) \leftarrow true$;
7: **for each** symbole d'entrée a **do**
8: $U \leftarrow \widehat{\varepsilon}(\delta(T,\ a))$;
9: **if** $U \notin E'$ **then**
10: Ajouter U à E';
11: Marqué$(U) \leftarrow false$;
12: **end if**
13: $\delta'(T,\ a) \leftarrow U$;
14: **end for**
15: **end while**
16: $F' \leftarrow \emptyset$;
17: **for each** $T \in E'$ **do**
18: **if** $T \cap F \neq \emptyset$ **then**
19: Ajouter T à F';
20: **end if**
21: **end for**

FIG. 2.14 – Algorithme de déterminisation d'un **AFND**.

2. $q'_0 = \widehat{\varepsilon}\{1\} = \{1,\ 2\}$.
3. • *Itération* 1 *:*
   $\delta'(\{1,\ 2\},\ a) = \widehat{\varepsilon}(\{1,\ 3\}) = \{1,\ 2,\ 3\}$ *;*
   $\delta'(\{1,\ 2\},\ b) = \widehat{\varepsilon}(\{3\}) = \{3\}$ *;*
   • *Itération* 2 *:*
   $\delta'(\{1,\ 2,\ 3\},\ a) = \widehat{\varepsilon}(\{1,\ 3\}) = \{1,\ 2,\ 3\}$ *;*
   $\delta'(\{1,\ 2,\ 3\},\ b) = \widehat{\varepsilon}(\{3\}) = \{3\}$ *;*
   $\delta'(\{3\},\ a) = \widehat{\varepsilon}(\{\emptyset\}) = \emptyset$ *;*
   $\delta'(\{3\},\ b) = \widehat{\varepsilon}(\{\emptyset\}) = \emptyset$ *;*
   On renomme les états : $\{1,\ 2\} = 1$, $\{1,\ 2,\ 3\} = 2$ *et* $\{3\} = 3$.
4. $F' = \{2,\ 3\}$.

*La représentation graphique de l'**AFD** obtenu est illustré par la figure suivante :*

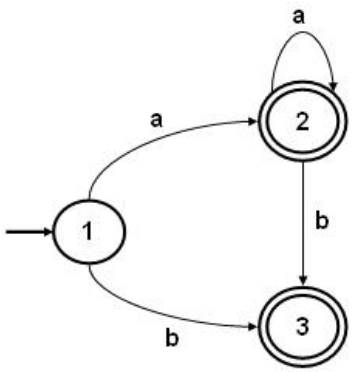

FIG. 2.15 – L'**AFD** obtenu par déterminisation de l'**AFND** de la figure 2.5.

2.10 Minimisation d'un AFD

Un **AFD** obtenu à partir d'un **AFND** peut avoir un nombre d'états plus grand que celui de l'**AFND**. D'un point de vue pratique, il est très intéressant d'avoir un **AFD** ayant un nombre minimal d'états. En théorie des langages, on montre que tout **AFD** possède un **AFD** équivalent minimal.

Nous présentons un algorithme de minimisation d'un **AFD** : l'**algorithme de Moore**. L'algorithme démarre avec un **AFD** simplifié (dont tous les états sont utiles). L'idée de l'**algorithme de Moore** est de **réduire** l'automate en identifiant les états appelés **inséparables**.

2.10.1 Simplification d'un AFD

Définition 2.10.1 *Soit M un **AFD**. Un état q de M est **accessible**, s'il existe au moins un chemin de l'état initial q_0 vers q.*

Définition 2.10.2 *Soit M un **AFD**. Un état q de M est **co-accessible**, s'il est accessible et s'il existe au moins un chemin menant de l'état q vers un état final q' de M.*

Définition 2.10.3 *Soit M un **AFD**. Un état q de M est **utile**, s'il est co-accessible. On dit que M est **simplifié**, s'il ne contient que des états utiles.*

On obtient un **AFD** simplifié en supprimant tous les états **non accessibles** et **non co-accessibles** ainsi que toutes les transitions qui s'y rapportent (sortantes et entrantes de ces états). Par exemple, simplifions l'**AFD** de la figure suivante :
L'**AFD** considéré n'est pas simplifié : l'état 7 est non accessible. On supprime cet état, ainsi que toutes les transitions qui s'y rapportent. On obtient l'**AFD** suivant :

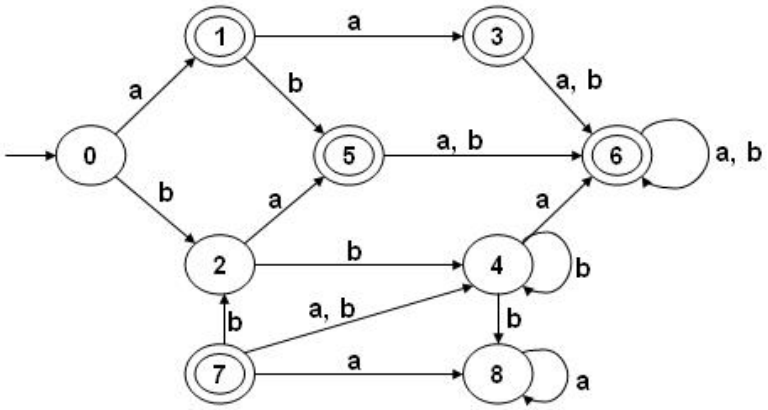

FIG. 2.16 – Un **AFD** à simplifier.

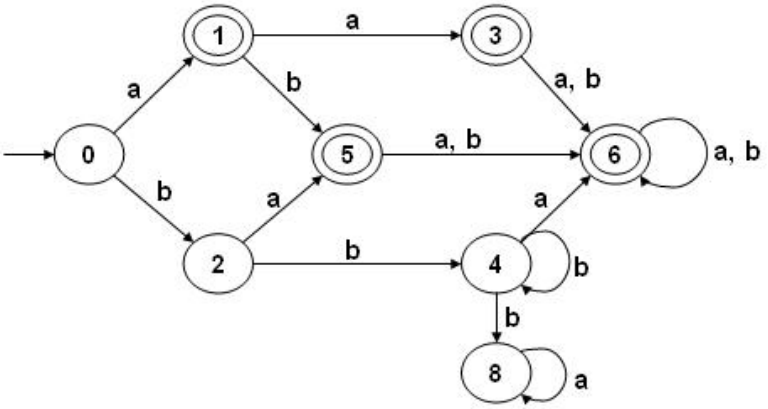

FIG. 2.17 – Suppression de l'état non accessibles 7.

On supprime l'état 8 car il n'est pas co-accessible, ainsi que toutes les transitions qui s'y rapportent. On obtient finalement l'**AFD** simplifié :

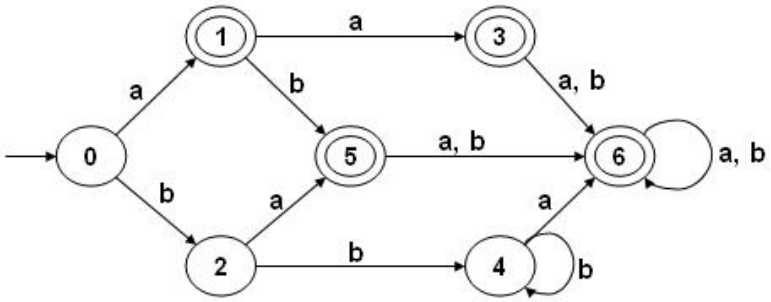

FIG. 2.18 – Un **AFD** à simplifié.

2.10.2 Équivalence de Nerode

Soit $M = (E,\ A,\ \delta,\ q_0,\ F)$ un **AFD** simplifié. Pour tout état $q \in E$, on note $L_q(M)$ le langage formé par tous les mots reconnus par l'automate M en prennant l'état q comme état initial. Formellement :

$$L_q(M) = \{w \in A^* \mid \delta^*(q,\ w) \in F\}$$

Lorque l'**AFD** M est fixé, on écrira L_q au lieu de $L_q(M)$.

Définition 2.10.4 *Deux états p et q sont dits **inséparables**, si et seulement si $L_p = L_q$. Dans le cas contraire, on dit que p et q sont **séparables**. Ainsi, p et q sont **séparables** si et seulement s'il existe un mot $w \in A^*$ tel que $\delta^*(p,\ w) \in F$ et $\delta^*(q,\ w) \notin F$ ou vice-versa. Si w est un mot vérifiant cette propriété, on dit qu'il **sépare** les états p et q.*

Exemple 2.10.1 *Dans l'automate de la figure 2.18, les états 3 et 6 sont inséparables puisque $L_3 = L_6 = (a|b)^*$, alors que les états 4 et 6 sont séparables puisque $L_4 = b^*a(a|b)^* \neq (a|b)^*$. Le mot ε sépare les états 4 et 6.*

Définition 2.10.5 *L'**équivalence de Nerode** sur M est la relation binaire $\sim$ définie sur l'ensemble des états E par $p \sim q$ si et seulement si p et q sont **inséparables**.*

Les algorithmes de minimisation des **AFD** se diffèrent de la manière de calculer les classes d'équivalence de cette relation $\sim$. Nous présentons un algorithme de minimisation très simple dû à **Hopcroft et Ullman** $[1979]$.

2.10.3 Algorithme de Hopcroft & Ullman

Soit M un **AFD** simplifié. Voici tout d'abord le pseudo-code de l'algorithme : L'idée de l'**algorithme de Hopcroft** $\&$ **Ullman** est de déterminer tous les groupes d'états qui peuvent être séparés par un mot d'entrée. Chaque groupe d'états qui ne peuvent pas être séparés est alors fusionné eu un état unique. L'algorithme fonctionne en mémorisant et en raffinant une partition de l'ensemble des états. Chaque groupe d'états à l'intérieur de la partition consiste en les états qui n'ont pas encore été séparés les uns des autres, et toute paire d'états extraits de différents groupes a été prouvé "séparés" par un mot.

Initialement, la partition consiste en geux groupes : les états finaux et les états non-finaux : $\Pi_0 = (F,\ E \setminus F)$. L'étape fondamentale consiste à prendre un groupe d'états, par exemple $G = \{e_1,\ e_2,\ ...,\ e_k\}$, et un symbole $a \in A$, et à examiner les transitions des états $\{e_1,\ e_2,\ ...,\ e_k\}$ sur a. Si ces transitions conduisent à des

Algorithm 4 Minimisation d'un **AFD** par la méthode de **Hopcroft** & **Ullman**

```
1: Partager E en deux groupes : F et (E \ F);
2: repeat
3:     for (each groupe G) do
4:         for (each symbole a) do
5:             for (each paire détats p et q dans G) do
6:                 if (δ(p, a) et δ(q, a) ne conduisent pas au
   même groupe) then
7:                     Partager G en deux sous-groupes : l'un
   contenant p et l'autre contenant q;
8:                 end if
9:             end for
10:        end for
11:    end for
12: until (aucune partition ne peut être produite);
```

FIG. 2.19 – Pseudo-code le l'algorithme de minimisation de **Hopcroft** & **Ullman**.

états qui tombent dans au moins deux groupes différents de la partition courante, alors on doit diviser le groupe G de manière que les transitions depuis chaque sous-ensemble de G soient toutes confiées à un seul groupe de la partition courante. Supposons par exemple que e_1 et e_2 mènent aux états t_1 et t_2 et que t_1 et t_2 soient dans des groupes différents de la partition. Alors, on doit diviser G en au moins deux sous-ensembles de telle manière qu'un sous-ensemble contienne e_1 et l'autre e_2. Notons que t_1 et t_2 sont séparés par un mot w, ce qui prouve que e_1 et e_2 sont séparés par le mot aw.

Nous répétons ce processus de division de groupes de la partition courante jusqu'à ce que plus aucun groupe n'ait besoin d'être divisé. Pour obtenir l'**AFD minimal**, on applique les règles suivantes :

- ▷ Transformer chaque **groupe** en un **état**.
- ▷ L'**état initial** est le **groupe qui contient l'état initial** de l'**AFD** de départ.
- ▷ Les **états finaux** sont les **groupes qui ne contiennet que des états finaux**

de l'**AFD** initial.

▷ La **matrice de transition** est obtenue en ne gardant qu'une seule ligne représentatrice d'un groupe dans la partition finale.

2.10.4 Un exemple d'application

Considérons l'**AFD** représenté par la figure suivante :

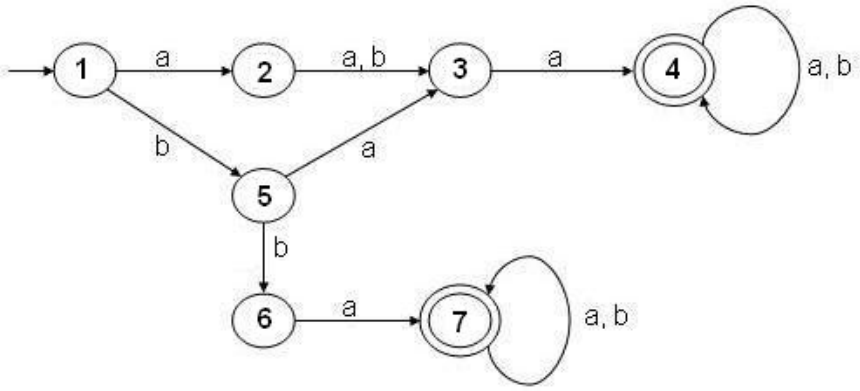

FIG. 2.20 – Un **AFD** simplifié à minimiser.

- La première partition est formée de deux groupes : $\#1 = \{1,\ 2,\ 3,\ 5,\ 6\}$ et $\#2 = \{4,\ 7\}$. On calcule la fonction de transition relative à cette partition :

Groupes	États	a	b
$\#1$	1	$\#1$	$\#1$
$\#1$	2	$\#1$	$\#1$
$\#1$	3	$\#2$	$\emptyset$
$\#1$	5	$\#1$	$\#1$
$\#1$	6	$\#2$	$\emptyset$
$\#2$	4	$\#2$	$\#2$
$\#2$	7	$\#2$	$\#2$

  Les états 1 et 3 sont séparables, ainsi que 1 et 6. Remarquons que le groupe $\#2 = \{4,\ 7\}$ est non cassable.

- On obtient alors une nouvelle partition avec trois goupes : $\#1 = \{1,\ 2,\ 5\}$, $\#2 = \{3,\ 6\}$ et $\#3 = \{4,\ 7\}$. On calcule la nouvelle fonction de transition relative à cette deuxième partition :

Groupes	États	a	b
$\#1$	1	$\#1$	$\#1$
$\#1$	2	$\#2$	$\#2$
$\#1$	5	$\#2$	$\#2$
$\#2$	3	$\#3$	$\emptyset$
$\#2$	6	$\#3$	$\emptyset$
$\#3$	4	$\#3$	$\#3$
$\#3$	7	$\#3$	$\#3$

Les états 1 et 2 sont séparables. Remarquons que le groupe $\#2 = \{3,\ 6\}$ est non cassable.

- On obtient alors une nouvelle partition avec quatre goupes : $\#1 = \{1\}$, $\#2 = \{2,\ 5\}$, $\#3 = \{3,\ 6\}$ et $\#4 = \{4,\ 7\}$. On calcule la nouvelle fonction de transition relative à cette partition :

Groupes	États	a	b
$\#1$	1	$\#2$	$\#2$
$\#2$	2	$\#3$	$\#3$
$\#2$	5	$\#3$	$\#3$
$\#3$	3	$\#4$	$\emptyset$
$\#3$	6	$\#4$	$\emptyset$
$\#4$	4	$\#4$	$\#4$
$\#4$	7	$\#4$	$\#4$

Le processus de partionnement est terminée car tous les groupes sont maintenant incassables.

- On obtient l'automate minimal en transformant chaque groupe en état. Il y a donc 4 états : $\#1$, $\#2$, $\#3$ et $\#4$. L'état initial est $q_0 = \#1$ et l'ensemble des états finaux est $F = \{\#4\}$. La fonction de transition est :

δ	a	b
$\#1$	$\#2$	$\#2$
$\#2$	$\#3$	$\#3$
$\#3$	$\#4$	
$\#4$	$\#4$	$\#4$

L'automate minimal trouvé est représenté par le graphe suivant :
Le langage reconnu par cet automate est alors : $(a|b)(a|b)a(a|b)^*$.

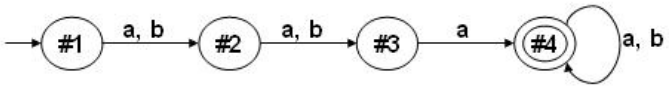

FIG. 2.21 – L'**AFD** minimal.

2.11 Langages réguliers et automates finis

D'après l'algorithme de **Thompson**, nous pouvons affirmer le résultat suivant (**Théorème de Thompson**) :

Théorème 2.11.1 *Soit A un alphabet. Si un langage L est **régulier** sur A, alors L est reconnu par un **AFD**.*

Le théorème suivant est dû à **Kleene** :

Théorème 2.11.2 *Soit A un alphabet. Un langage L est **régulier** sur A, si et seulement s'il est reconnu par un **AFD**.*

Pour démontrer ce théorème, il suffit de prouver que si un langage L est réconnu par un **AFD**, alors il est régulier, ce qui revient à exhiber pour tout **AFD** une expression régulière équivalente. Il existe plusieurs algorithmes pour obtenir cette expression. Nous présentons premièrement une méthode appélée l'algorithme **BMC** (***Brzozowski*** et ***McCluskey***).

2.11.1 Algorithme BMC

Soit $M = (E,\ A,\ \delta,\ q_0,\ F)$ un **AFD**. On cherche une expression régulière dénotant le langage reconnu par M. On va procéder par suppression de transitions et d'états, en remplaçant d'autres étiquettes par des expressions régulières :

1. Ajouter à M deux nouveaux états, notés α (état initial) et ω (état final), et les transitions $(\alpha,\ \varepsilon,\ q_0)$ et $(q_f,\ \varepsilon,\ \omega)$ pour tout état $q_f \in F$.
2. Répéter les actions suivantes tant que possible :

   - S'il existe deux transitions $(p,\ r,\ q)$ et $(p,\ s,\ q)$, les remplacer par la transition $(p,\ r|s,\ q)$.

   - Supprimer un état q (autre que α et ω) et remplacer, pour tout états $p,\ r \neq q$, les transitions $(p,\ s,\ q)$, $(q,\ t,\ q)$ et $(q,\ u,\ r)$, par la transition $(p,\ st^*u,\ r)$.

Cet algorithme s'arrête toujours, parce que l'on diminue, à chaque itération, le nombre de transitions et d'états, jusqu'à obtenir une seule transition $(\alpha,\ e,\ \omega)$. Il est clair que e est une expression régulière pour le langage $L(M)$ reconnu par M.

Comme exemple, considérons l'**AFD** de la figure 2.5. On lui ajoute les états α et ω, et on obtient l'automate suivant :

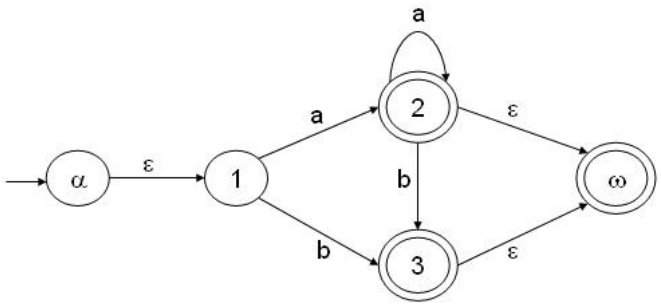

FIG. 2.22 – L'automate de la figure 2.5, augmenté de deux états α et ω.

▷ Supprimons l'état 2. Les couples de transitions concernées sont $(1,\ a,\ 2)$ et $(2,\ \varepsilon,\ \omega)$ d'une part, et $(1,\ a,\ 2)$ et $(2,\ b,\ 3)$ d'autre part. Le premier couple produit une transition $(1,\ aa^* = a^+,\ \omega)$, le deuxième produit une transition $(1,\ aa^*b = a^+b,\ 3)$, ce qui donne l'automate :

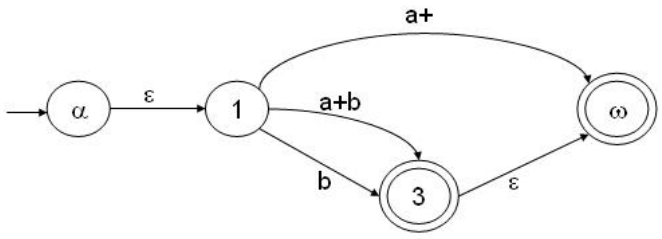

FIG. 2.23 – L'automate précédent, après suppression de l'état 2.

▷ Supprimons les deux transitions $(1,\ a^+b,\ 3)$ et $(1,\ b,\ 3)$, et remplaçons les par la transition $(1,\ a^+b \mid b = a^*b,\ 3)$. Celà donne l'automate :

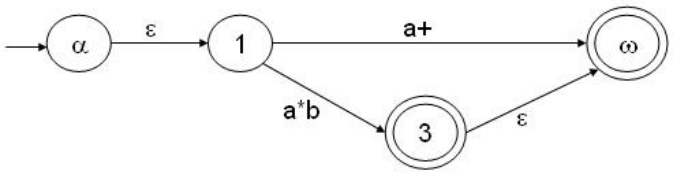

FIG. 2.24 – L'automate précédent, après suppression des deux transitions $(1,\ a^+b,\ 3)$ et $(1,\ b,\ 3)$.

▷ Supprimons l'état 3, le couple de transitions concernées est $(1,\ a^*b,\ 3)$ et $(3,\ \varepsilon,\ \omega)$, qui produit une transition $(1,\ a^*b,\ \omega)$. Ce qui donne l'automate :

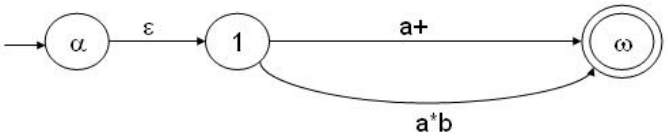

FIG. 2.25 – L'automate précédent, après suppression de l'état 3.

▷ Supprimons les deux transitions $(1,\ a^+,\ \omega)$ et $(1,\ a^*b,\ \omega)$, et remplaçons les par la transition $(1,\ a^+ \mid a^*b = a^*(a \mid b),\ \omega)$. Celà donne l'automate :

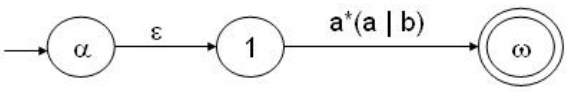

FIG. 2.26 – L'automate précédent, après suppression des deux transitions $(1,\ a^+,\ \omega)$ et $(1,\ a^*b,\ \omega)$.

▷ Il ne reste plus qu'à supprimer l'état 1. On obtient l'automate suivant :

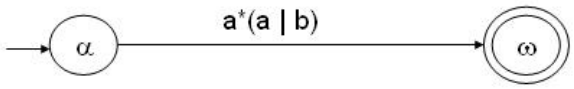

FIG. 2.27 – L'automate complètement réduit.

▷ Finalment, on obtient l'expression régulière $a^*(a|b)$.

2.11.2 Méthode algébrique

À partir d'un **AFD** $M = (E,\ A,\ \delta,\ q_0,\ F)$, on va construire un système de n équations linéaires (n étant le nombre des états de M). Les inconnus de ce système sont les langages

$$X_q = \{w \in A^* \mid \delta^*(q,\ w) \in F\}$$

où $q \in E$ est un état de l'automate. X_q est le langage formé de tous les mots reconnus par M, en prenant q comme état initial. Nous avons les résultats suivants :

Lemme 2.11.1 *Soient p et q des états d'un **AFD** $M = (E,\ A,\ \delta,\ q_0,\ F)$.*

1. *Si $\delta(p,\ a) = q$, alors $aX_q \subseteq X_p$.*
2. *Si $\delta(p,\ a) = p$, alors $aX_p \subseteq X_p$.*
3. *Si $q \in F$, alors $\varepsilon \in X_q$.*

Le système d'équations linéaires est obtenu en appliquant ces règles. Voici un exemple de ce système pour l'automate de la figure 2.5 :

$$\begin{cases} X_1 = aX_2 \mid bX_3 \\ X_2 = aX_2 \mid bX_3 \mid \varepsilon \\ X_3 = \varepsilon \end{cases} \tag{2.3}$$

Un tel système peut être résolu facilement en appliquant la méthode des substitutions et le résultat du lemme suivant :

Lemme 2.11.2 (*Lemme d'**Arden***). *Soient E et F deux langages sur un alphabet A. Si $\varepsilon \notin E$, alors l'unique solution de l'équation $X = EX \mid F$ est $X = E^*F$.*

Preuve du lemme d'Arden. Posons $Y = E^*F$. Alors, $EY \mid F = EE^*F \mid F = E^+F \mid F = (E^+ \cup \{\varepsilon\})F = E^*F = Y$, ce qui montre que E^*F est bien une solution de l'équation $X = EX \mid F$.

Supposons maintenant que l'équation $X = EX \mid F$ possède au moins deux solutions L et K. On a : $L \setminus K = (EL \mid F) \setminus K = EL \setminus K$, car $F \subseteq K$. Or, $(EL \setminus K) \subseteq (EL \setminus EK)$, puisque $EK \subseteq K$. Par suite $(L \setminus K) \subseteq E(L \setminus K)$, car $(EL \setminus EK) \subseteq E(L \setminus K)$. Par récurrence, on obtient que $(L \setminus K) \subseteq E^n(L \setminus K)$, pour tout entier $n \geq 1$. Comme le mot vide ε n'appartient pas à E, alors tout mot de $L \setminus K$ a pour longueur au moins n pour tout $n \geq 1$. Par conséquent, $L \setminus K = \emptyset$ et alors $L \subseteq K$. On montre de même que $K \subseteq L$ (car L et K jouent des rôles syméetriques). Finalement, on a montré que $L = K$.

Résolvons le système (2.3). En substituant la troisième équation dans les deux premières, on obtient :

$$\begin{cases} X_1 = aX_2 \mid b \\ X_2 = aX_2 \mid (b \mid \varepsilon) \\ X_3 = \varepsilon \end{cases} \tag{2.4}$$

En appliquant le lemme d'Arden à la deuxième équation, on trouve :

$$X_2 = a^*(b|\varepsilon)$$

Finalement, on obtient :

$$X_1 = a(a^*(b|\varepsilon))|b = a^+b|a^+|b = (a^+|\varepsilon)b|a^+ = a^*b|a^*a = a^*(a|b)$$

2.12 Lemme de pompage

Étant donné un alphabet A et un langage L défini sur A. Comment savoir si L est régulier ? Pour une réponse affirmative, il suffit de trouver une expression régulière qui le dénote ou un automate fini qui le reconnaît. Pour une réponse négative, on utilise un outil théorique : le lemme de **pompage**.

Théorème 2.12.1 *Si un lanage L est régulier sur A, alors il existe un entier naturel N tel que pour tout mot $w \in L$, de longueur $|w| \geq N$, w s'écrit sous la forme $w = xyz$ avec :*

1. $|xy| \leq N$.
2. $y \neq \varepsilon$, (*ou bien* $|y| \geq 1$).
3. $xy^kz \in L, \forall k \geq 0$.

Intuitivement, N c'est le nombre des états de l'**AFD** minimal reconnaissant L. Pour accépeter un mot $w \in L$ de longueur $|w| \geq N$, il faut au moins visiter un

certain état e plus d'une fois (ce qui est illustré par la figure 2.29). Soit le mot x lu depuis l'état initial q_0 jusqu'à l'état e, le mot y lu sur une boucle autour de l'état e et le mot z lu depuis l'état e vers un état final q_f. Il est clair alors que $w = xyz$ et que l'on a : $|xy| \leq N$, $y \neq \varepsilon$ et $xy^k z \in L, \forall k \geq 0$.

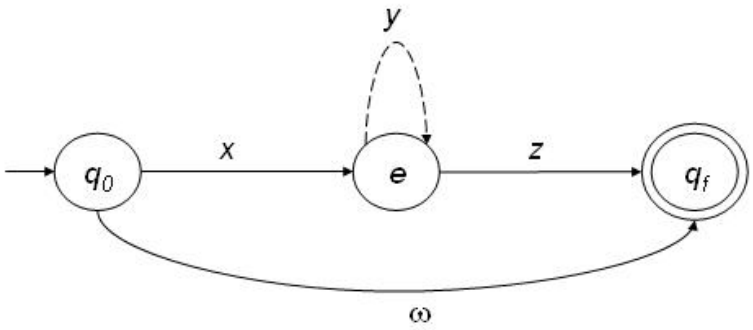

FIG. 2.28 – Automate minimal et lemme de pompage.

Exemple 2.12.1 *Le langage* $L = \{a^n b^n \mid n \geq 0\}$ *sur* $A = \{a,\ b\}$ *n'est pas régulier.*

Preuve. On utilise le lemme de pompage. Supposons que L est régulier. Il vérifie le lemme de pompage. Notons N l'entier assuré par ce lemme. Prenons le mot $w = a^N b^N$ et soit une décomposition w de la forme $w = xyz$. Comme $|xy| \leq N$ et $|y| \geq 1$, alors $x = a^i$, $y = a^j$ et $z = a^{N-i-j} b^N$, avec $i \geq 0$ et $j \geq 1$. Le mot $xy^0 z = a^{N-j} b^N \in L$, ce qui oblige que $j = 0$. Ceci est impossible, puisque $j \geq 1$, ce qui entraîne une contraduction. Le langagee L n'est pas alors régulier.

Chapitre 3

Alanyse lexicale

3.1 Rôle de l'analsye lexicale

Le programme d'**analyse lexicale** (*scanning*) est assuré généralement par un module appelé **analyseur lexical** (*scanner*). Le rôle principal de l'analyseur lexical est de combiner les caractères du fichier source pour former des mots ayant un sens pour le langage source. Ces mots sont appelés **lexèmes** (ou *token*). Au passage, cette phase doit :

- ▷ reconnaître les mots réservés, les constantes, les identificateurs, ...etc.
- ▷ signaler les erreures lexicales (mots mal orthographiés) et relier les messages d'erreurs issues du compilateur au programme source.
- ▷ ignorer les commentaires et les caractères blancs.
- ▷ effectuer, si nécessaire, un pré-traitement du texte source.

3.2 Unités lexicales, lexèmes et modèles

3.2.1 Definitions

Définition 3.2.1 *Une **unité lexicale** représente une classe particulière du lexique (vocabulaire) du langage source.*

Exemple 3.2.1 *Les **mots-clés** et les **identificateurs** sont deux **unités lexicales**.*

Définition 3.2.2 *Un **lexème** (ou **token**) est une chaîne de caractères qui appartient à une unité lexicale.*

Exemple 3.2.2 *En langage C, **if**, **for**, **int** et **return** sont des **lexèmes** de l'unité lexicale **mot-clé**.*

Définition 3.2.3 *Un **modèle** d'unité lexicale est une règle qui décrit l'ensemble des lexèmes qui seront des instances de cette unité lexicale.*

Exemple 3.2.3 *En langage PHP, le **modèle** d'un identificateur est une chaîne de caractères qui commence par une lettre (minuscule ou majuscule) ou le caractère '_' ou le caractère '$' et qui ne contient que des lettres, des chiffres et des caractères '_' et '$'.*

3.2.2 Les unités lexicales les plus courantes

La plupart des langages de programmation disposent des unités lexicales suivantes :

- ▷ Les **mots-clés** (*keywords*) : leurs définitions et leurs utilisations sont fixées par le langage. Exemples de mots-clés dans C : `if`, `else`, `for`, `while`, `do`, `switch`, ...etc.
- ▷ Les **identificateurs** : ce sont les noms qu'un programmeur utilise pour désigner les entités d'un programme (variables, constantes, types, fonctions).
- ▷ Les **opérateurs** : symboles d'opérations arithmétiques, logiques et de comparaison.
- ▷ Les **nombres** : entiers ou réels, positifs ou négatifs.
- ▷ Les **chaînes de caractères** : séqeunces de caractères alphanumériques.
- ▷ Les **délimiteurs** : espace, tabulation et retour à la ligne.
- ▷ Les **symboles spéciaux** : ( ) [ ] { } ...etc.

3.3 Interface avec l'analyseur syntaxique

3.3.1 Position de l'analyseur lexical

Le plus souvent, l'analyseur lexical est implementé comme un sous-programme de l'analyse syntaxique. Ce dernier ordonne à l'analyseur lexical de délivrer la prochaine unité lexicale du programme source. L'analyseur lexical lit alors la suite

des caractères non encore lus et délivre la première unité lexicale rencontrée. La figure ci-dessous montre la relation (de type producteur/consommateur) qui relie l'analyseur lexical et l'analyseur syntaxique :

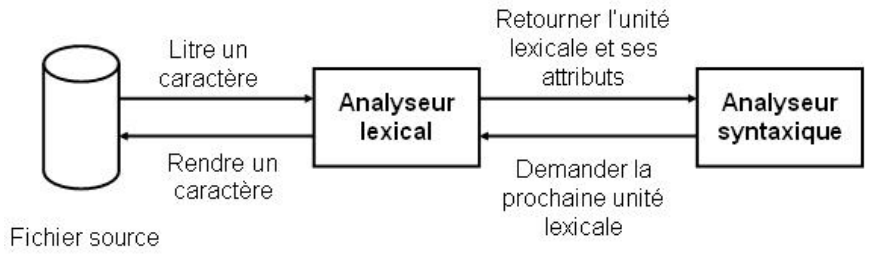

FIG. 3.1 – Relation entre l'analyseur lexical et l'analyseur syntaxique.

3.3.2 Attribut d'un lexème et table des symnoles

Savoir à quelle unité lexicale est associé un lexème est essentiel pour l'analyse syntaxique. Cependant, les étapes suivantes de la compilation auront besoin d'autres informations sur les lexèmes. L'analyseur lexical doit alors réunir les informations utiles sur les lexèmes. Ces informations sont stockées dans une table, appelée **table des symboles**, sous forme d'**attributs**. Un attribut pour un lexème peut être sa valeur, sa position dans le fichier source, ...etc.

3.4 Implémentation d'un analyseur lexical

Il y a trois méthodes classiques pour coder un analyseur lexicale :

- La méthode "**manuelle**" : l'analyseur lexical est écrit en chair et en os à la main en utilisant un langage évolué (comme C, $Pascal$, ou $Java$).
- La méthode "**semi-automatique**" : on simule l'**AFD** (ou les **AFD**) qui reconnaît les unités lexicales en utilisant un langage de haut-niveau. L'**AFD** étant construit à la main.
- La méthode "**automatique**" : l'analyseur lexical est généré à partir d'une description par un outil informatique qui génère automatiquement le code exécutable de l'analyseur. Cette méthode sera présenteé dans le chapitre suivant (L'outil **Flex**).

3.5 La méthode manuelle

L'écriture d'un analyseur lexical à la main n'est pas souvent très difficile : on ignore tous les caractères inutiles, jusqu'à ce qu'on trouve le premier caractère significatif. En suite, on teste si le prochain lexème est un mot-clé, un nombre, un identificateur, ...etc.

On va maintenant implémenter dans un langage de haut-niveau (langage C, par exemple) un analyseur lexical qui lit et convertit le flot d'entrée en un flot d'unités lexicales. Pour simplifier, nous nous limitons à un fragment d'un langage source fictif. Pour ce fragment du langage considéré, l'analyseur lexical reconnaîtra les unités lexicales suivantes :

- ▷ **KEYWORD** qui dénote les quatre mots-clés : **si**, **alors**, **sinon** et **finsi**.
- ▷ **IDENT** qui dénote les identificateurs.
- ▷ **NBENT** qui dénote les nombres entiers. L'attribut est dans ce cas, la valeur décimale de l'entier reconnu.

On fait également les hypothèses suivantes :

- les mots-clés sont réservés, c'est-à-dire qu'ils ne peuvent pas être utilisés comme identificateurs.
- les lexèmes sont séparés par un caractère blanc (espace, tabulation et retour à la ligne). Notre analyseur éliminera ces caractères.

La figure suivante suggère la manière dont l'analyseur lexical, représenté par la fonction `AnalLex()`, réalise sa tâche. La fonction `getchar()` permet de lire le prochain caractère sur l'entrée standard et la fonction `ungetc()` permet de rendre ce caractère au flot d'entrée.

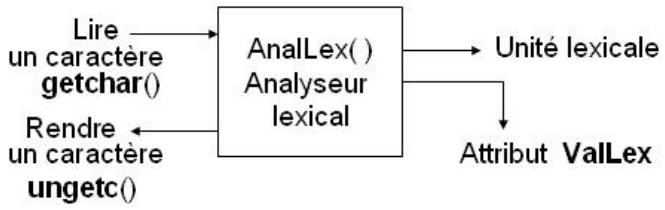

FIG. 3.2 – Implémentation d'un anlyseur lexical écrit à la main (en langage C).

Ainsi, le code suivant :

```
carlu = gectchar(); /* carlu : variable de type char */
ungetc(carlu, stdin);
```

laisse le flot d'entrée inchangé. La première instruction affecte le prochain caractère de l'entrée à la variable `carlu`, alors que la deuxième rend à l'entrée standard *stdin* la valeur de la variable `carlu`.

La fonction `AnalLex`() rend un entier qui code une unité lexicale. L'unité lexicale, comme **KEYWORD**, peut alors être codé par un entier supérieur aux entiers encodant les caractères, par exemple 256. Pour permettre un changement aisé de l'encodage, on utilise une constante symbolique `KEYWORD` pour désigner l'entier enocdant **KEYWORD**. Pour réaliser ceci en C, on utilise une directive de définition :

```
#define KEYWORD 256
#define IDENT   257
#define NBENT   258
```

Pour envoyer l'attribut d'un lexème, on utilise une variable globale `ValLex`. Cette variable est déclarée comme **union** :

```
union ValLex
{
   int  valint; /* pour un entier */
   char valstr[33];
/* pour un identificateur ou un mot clé */
};
```

La fonction `EstMotCle`() teste si un identificateur est un mot-clé. La fonction `Erreur` affiche un message d'erreur. L'analyseur ignore les caractères blancs (espaces, tabulations et retour à la ligne). S'il reconntre une chaîne non spécifiée et qui n'est pas ignoré, il signale une erreur et quite l'analyse lexicale. La figure 3.3 présente le pseudo-code de notre analyseur écrit à la main.

Algorithm 3 Pseudo-code de l'analyseur

```
1: while (Vrai) do
2:     carlu := getchar();
3:     if (carlu est un caractre blanc) then
4:         Ne rien faire;
5:     else
6:         if (carlu est un chiffre) then
7:             Initialiser ValLex;
8:             carlu := getchar();
9:             while (carlu est un chiffre) do
10:                Mettre à jour ValLex;
11:                carlu := getchar();
12:            end while
13:            ungetc(carlu, stdin);
14:            return NBENT;
15:        else
16:            if (carlu est une lettre ou '_)' then
17:                Initialiser ValLex;
18:                while (isalnum(carlu) ou '_') do
19:                    Mettre à jour ValLex;
20:                    carlu := getchar();
21:                end while
22:                ungetc(carlu, stdin);
23:                if (ValLex est un mot-clé) then
24:                    return KEYWORD;
25:                else
26:                    return IDENT;
27:                end if
28:            end if
29:        end if
30:    end if
31: end while
```

FIG. 3.3 – Pseudo-code de l'anlyseur lexical.

3.6 La méthode semi-automatique

Les **AFD** constituent un outil important pour l'écriture d'analyseurs lexicaux, car leur implémentation avec un langage évolué (de haut-niveau) est très facile. Si nous disposons d'un **AFD** qui reconnaît les unités lexicales du langage source,

nous pouvons effectuer l'analyse lexicale de ce langage en écrivant un programme qui simule le travail de l'**AFD**.

Pour notre cas, la figure 3.4 illustre un **AFD** qui accèpte les unités lexicales du fragment du langage étudié.

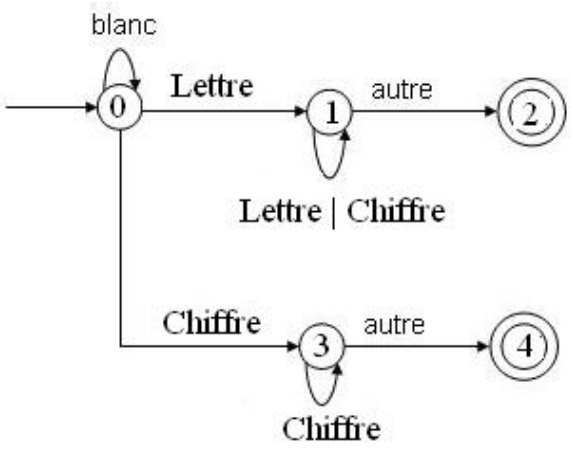

FIG. 3.4 – Un **AFD** qui reconnaît le fragment du langage étudié.

On part de l'état 0, car c'est l'état initial de l'**AFD**. On lit un caractère sur l'entrée, puis on effectue une séquence de test :

- Si le caractère lu est un symbole blanc (espace, tabulation ou retour à la ligne), on reste dans l'état 0.

- Si le caractère lu est une lettre, on transite à l'état 1. C'est la première lettre d'un identificateur ou d'un mot-clé. On initialise le premier caractère de la chaîne à ce caractère lu.

- Si le caractère lu est un chiffre, on transite à l'état 3. C'est le premier chiffre d'un nombre. On initialise la valeur du nombre avec la valeur décimale de ce caractère.

- Si le caractère lu est **EOF** (le caractère de fin de fichier), on quite l'analyse lexicale.

- Pour tout autre caractère lu, signaler une erreur lexicale et sortir.

Lorsqu'on arrive à l'état 1, on lit un caractère, puis on effectue les tests suivants :

1. Si le caractère lu est une lettre ou un chiffre, on reste dans l'état 1, tout en enregistrant le caractère lu dans la position adéquate de la chaîne.

2. Pour tout autre caractère lu, on rend le caractère lu et l'on passe à l'état 2.

Lorsqu'on arrive à l'état 3, on lit un caractère, puis on effectue le test suivant :

1. Si le caractère lu est un chiffre, on reste dans l'état 3, tout en continuant l'évaluation du nombre.

2. Pour tout autre caractère lu, on rend le caractère et l'on passe à l'état 4.

Le traitement des états finaux s'effectue comme suit :

1. Si l'état courant est 2, reculer la lecture du caractère. Si la chaîne trouvée est un identificateur, retourner **IDENT**, sinon retourner **KEYWORD**.

2. Si l'état courant est 4, reculer la lecture du caractère et retourner **NBENT**.

Voici le pseudo-code de la simulation de l'**AFD** de notre analyseur :

Algorithm 4 Simulation de l'**AFD**

```
1: Etat := 0;
2: repeat
3:     switch(Etat)
4:     Cas 0 : carlu = getchar();
5:     if (carlu est un blanc) then
6:         Etat = 0;
7:     else
8:         if (carlu est un chiffre) then
9:             Initialiser ValLex; Etat := 3;
10:        else
11:            if (carlu est une lettre ou '_') then
12:                Initialiser ValLex; Etat := 1;
13:            end if
14:        end if
15:    end if
16:    Cas 1 : carlu = getchar();
17:    if (isalnum(carlu) ou '_') then
18:        Etat = 1; Mettre à jour ValLex;
19:    else
20:        ungetc(carlu, stdin); Etat := 2;
21:    end if
22:    Cas 2 :
23:    if (ValLex est un mot-clé) then return KEYWORD;
24:    else return IDENT;
25:    end if
26:    Cas 3 : carlu = getchar();
27:    if (carlu) est numrique) then
28:        Etat = 3; Mettre à jour ValLex;
29:    else
30:        ungetc(carlu, stdin); Etat := 4;
31:    end if
32:    Cas 4 : return NBENT;
33: until (carlu == EOF)
```

FIG. 3.5 – Pseudo-code du simulateur de l'**AFD** de l'anlyseur lexical.

Chapitre 4

L'outil Flex

4.1 Présentation de Flex

4.1.1 Le compilateur Flex

Flex (***Fast Lexer***) est un **générateur d'analyseurs lexicaux**. **Flex** prends en entrée un fichier source (".*lex*" ou ".*flex*") contenant une description d'un analyseur lexical et délivre un fichier nommé "$lex.yy.c$" qui contient le code C du future analyseur lexical. Ce dernier, lorsqu'il est compilé avec `gcc` produira l'exécutable de l'analyseur lexical. La compilation d'un fichier **Flex**, s'effectue via la commande **flex**. Voici le schéma à suivre pour construire un analyseur lexical avec le compilateur **Flex** :

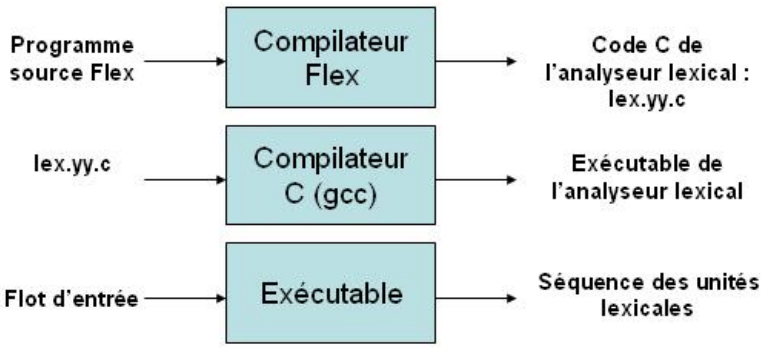

FIG. 4.1 – Création d'un analyseur lexical avec **Flex**.

Le fichier de description contient princialement des **expressions régulières** et des **actions** à exécuter lors de la reconnaissance de chaque lexème. **Flex** prends cette description et génère un **AFND** combiné (réunion de tous les **AFND** de l'ensemble des expressions régulières décrites). Ensuite **Flex** convertit cet **AFND** en

un **AFD**, puis le minimise et génère le code C qui va simuler l'automate minimal.

4.1.2 La fonction yylex

Le fichier "$lex.yy.c$" fournit une fonction externe nommée "**yylex**$()$" qui va réaliser l'analyse lexicale des lexèmes. La compilation de ce fichier par **gcc**, et sa liaison avec la librairie "**lfl**" de **Flex** produira un **analyseur lexical**. Par défaut, ce derniers va lire les données à partir de l'entrée standard $(stdin)$ et écrira les résultats sur la sortie standrad $(stdout)$. La fonction **yylex** n'a pas d'argument et retourne un **entier** qui désigne le **code de l'unité lexicale reconnue**.

4.2 Format d'un programme Flex

Flex est crée pour être utilisé avec C (ou C++). Un **programme Flex** se compose de trois **sections** (dans cet ordre) : la secdion des **définitions**, la section des **règles**, et la section du **code auxilliaire**. Voici le format d'un fichier **Flex** :

```
%{
   /* Inclusions et Déclarations */
%}
   /* Définitions des expressions régulières */
%%
   /* Règles */
%%
   /* Code C auxilliaire */
```

- La section des **définitions** comprend : les inclusions de fichiers, les déclarations globales C (variables, types et fonctions) et les définitions des expressions régulières.

- La section des **règles** contient les expressions régulières qui dénoteront les lexèmes à reconnaître et les actions à exécuter.

- La section du **code auxilliaire** contient les routines C nécessaires pour le fonctionnement désiré de l'analyseur.

Les déclarations sont copiées en haut dans le fichier "$lex.yy.c$", alors que les routines sont copiées en bas. Les définitions permettent de donner des noms à des expressions régulières. Ceci a pour objectif de simplifier les écritures des expressions régulières complexes dans la section des règles et d'augmenter leur lisibilité et leur identification.

4.3 Les règles

Une **règle** est formée d'une **expression régulière** (appelée **motif**) suivie d'une **séquence d'instructions** C (nommée **action**). L'idée est que chaque fois que l'analyseur lit une entrée (une chaîne de caractèes) qui vérifie le motif d'une règle, il exécute l'action associée avec ce motif. Par exemple, si l'on veut afficher un identificateur C, on écrira la règle suivante :

```
[a-zA-Z_][a-zA-Z_0-9]*  printf("ID : %s\n", yytext);
```

Les **règles** doivent être écrites après le premier symbole $\%\%$. Les **motifs** doivent commencer en colonne 0 et sont écrits sur une seule ligne. Les **actions** doivent commencer sur la même ligne, mais peuvent tenir sur plusieurs lignes. Voici la forme de la section des règles :

```
exp_reg_1 action_1
exp_reg_2 action_2
...
exp_reg_n action_n
```

4.4 Variables et fonctions prédéfinies

- ▷ **yytext** : (`char`$*$) chaîne de caractères qui contient le lexème courant reconnu.
- ▷ **yyleng** : (`int`) longueur de yytext (aussi de ce lexème).
- ▷ **yylineno** : (`int`) numéro de la ligne courante.
- ▷ **yyin** : (`File`$*$) fichier d'entrée à partir duquel **Flex** va lire les données à analyser. Par défaut, `yyin` pointe vers `stdin`.
- ▷ **yyout** : (`File`$*$) fichier de sortie dans lequel **Flex** va écrire les résultats de l'analyse. Par défaut, `yyout` pointe vers `stdin`.
- ▷ **unputc**(car) : fonction qui remet le caractère car dans le flot d'entrée.
- ▷ **main**( ) : la fonction `main` par défaut contient juste un appel à la fonction **yylex**. Elle peut être redéfinie par l'utilisateur dans la section des procédures auxiliaires.

4.5 Routines de Flex

4.5.1 Règles pour les actions

- Si pour un **motif**, l'**action n'est pas présente**, alors le lexème correspondant sera tout simplement ignoré. La même chose se produira si l'on a spécifier l'action notée ';'. Par exemple, voici un programme qui supprimera toutes les occurrences des mots "mauvais" et "bon" :

```
%%
"bon"      /* l'action est absente */
"mauvais"  ;
```

- **Flex** recopie dans le fichier de sortie toutes les chaînes de l'entrée non reconnues par aucune expression règluière : c'est l'**action par défaut**.

- Une action "|" signifie "**la même chose que l'action de la règle suivante**". Elle est utilisée quand plusieurs règles partagent la même action. Voici un exmple :

```
%%
a    |
ab   |
abc  |
abcd printf("a, ab, abc ou abcd\n");
```

4.5.2 Directives de Flex

Flex offre un nombre de **routines spéciales** qui peuvent être intégrées avec les actions :

- **ECHO** : copie le contenu de la variable **yytext** dans le fichier de sortie. Si ce dernier est **stdout**, `ECHO` est équivalente à l'action
  `printf("\%s", yytext);`

- **REJECT** : force l'analyseur à se **brancher à la deuxième meilleure règle applicable**. Par exemple, voici un exemple qui compte tous les mots d'un fichier, sauf les mots "bon" :

```
%{
   int nb_mots = 0;
%}
%%
bon     REJECT;
[a-z]*  nb_mots++;
```

▷ **yymore**() : demande à l'analyseur de **concaténer le contenu du lexème courant au lieu de le remplacer**. Par exemple, étant donnée l'entrée "mega-octet", le programme suivant produira l'affichage "mega-mega-octet" :

```
%%
mega- ECHO; yymore();
octet ECHO;
```

▷ **yyless**(n) : **rend tout les caractères du lexème courant dans le flot d'entrée**, **sauf les** n **premiers caractères de ce lexème**. Les variables **yytext** et **yyleng** seront ajustées automatiquement. Par example, sur l'entrée "foobar", le programme suivant affichera "foobarbar" :

```
%%
foobar  ECHO; yyless(3);
[a-z]+  ECHO;
```

▷ **yyterminate**() : **termine l'exécution de l'analyseur** et **renvoie la valeur** 0 **à l'appelant de l'analyseur**, en indiquant le message "$all\ done$". Par défaut, 'yyterminate()' est appelée lorsqu'un caractère **EOF** est lu. Par example, le programme suivant affichera le premier entier rencontré dans un fichier puis résumera l'analyseur :

```
%%
[0-9]+  ECHO; yyterminate();
```

4.6 Comment Flex génère un analyseur lexical

La table des transitions de l'**AFD** généré par le compilateur **Flex** est parcourue dans la routine **yylex**() par le code qui anime l'automate. Ce code est invariable

et est généré automatiquement pour la routine **yylex**(). La figure 4.2 illustre le pseudo-code de ce programme de simulation de l'automate.

Algorithm 5 Pseudo-code de simulation de l'AFD de Flex

```
1: repeat
2:     Etat_cur := Etat_init;
3:     carlu := getchar();
4:     while (transit[Etat_cur][carlu]) defini) do
5:         Etat_cur := transit[Etat_cur][carlu];
6:         yytext := strcat(yytext, carlu);
7:         yyleng++;
8:         if (EstFinal(Etat_cur)) then
9:             dernier_Etat_final := Etat_cur;
10:            long_finale := yyleng;
11:            carlu := getchar();
12:        end if ▷ Reculer jusqu'au dernier état final atteint
13:    end while
14:    Switch(Expr_reg[dernier_Etat_final])
15:      Cas 1 : Code_action_1
16:      ...
17:      Cas n : Code_action_n
18:    endSwitch
19: until Fin de fichier
```

FIG. 4.2 – Programme de simulation de l'**AFD** généré par **Flex**.

4.7 Situation à l'arrêt de l'automate

Lorsque plusieurs expressions réguliès peuvent s'appliquer pour analyser le début de la chaîne d'entrée, les règles suivantes sont appliquées à la construction de l'automate pour enlever l'ambiguïté entre les règles :

- ▷ La règle conduisant à la reconnaissance de la plus longue est d'abord choisie.
- ▷ A égalité de longueur reconnue, c'est la première règle qui est choisie.

Si aucun état final n'a été atteint (aucune expression ne reconnaît l'entrée) l'action par défaut est déclenchée (recopie du caractère d'entrée sur la sortie).

4.8 Exemple d'un analyseur lexical avec Flex

Pour terminer, nous donnons le code **Flex** d'un analyseur lexical du fragment du langage source étudié dans le chapitre précédent (chapitre 3). Cet analyseur affichera les lexèmes et leur unités lexicales. Tout d'abord, on écrit le code **Flex** dans un fichier texte enregistré avec l'extension "lex", disons "lexer.lex" :

```
%{
#include <stdlib.h>
%}
%%
alors |
finsi |
si    |
sinon printf("%s : KEYWORD\n", yytext);
[a-zA-Z_][a-zA-Z_0-9]* printf("%s : IDENT\n", yytext);
[0-9]+  printf("%d : NBENT\n", atoi(yytext))d;
[ \t\n] ;
```

Ensuite, nous compilons ce fichier pour obtenir l'exécutable de notre analyseur, disons "lexer.exe" :

FIG. 4.3 – Étapes de compilation d'un fichier **Flex**.

Supposons que l'on veut analyser le fichier texte suivant, disons "data.txt" :

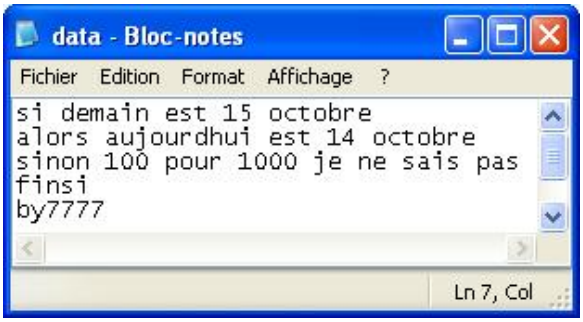

FIG. 4.4 – Fichier source à analyser.

Voici le résultat obtenu :

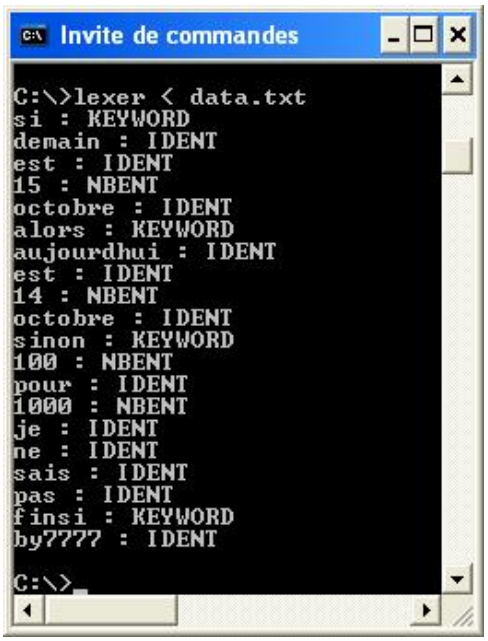

FIG. 4.5 – Résultat de l'analyse lexicale.

4.9 Quelques options de compilation avec Flex

- $-d$: exécute l'analyseur en mode de débougage.
- $-s$: supprime l'effet de la règle par défaut de **Flex**.

- ▷ $-t$: écrit le code de l'analyseur généré sur la sortie standard, au lieu de l'écrire dans le fichier "$lex.yy.c$".
- ▷ $-T$: affiche les informations sur l'automate fini déterministe crée par **Flex**.
- ▷ $-V$: affiche la version de **Flex** utilisé.

Chapitre 5

Grammaires hors-contexte

Dans le chapitre 2, on a vu deux **formalismes** équivalents pour **spécifier et reconnaître des unités lexicales** : les "**expressions régulières**" et les "**automates finis**". Malheurement, ces deux formalismes ne peuvent pas décrire certains langages. La théorie des langages montre par exemple qu'il est impossible de décrire le langage $\{a^n b^n \mid n \geq 0\}$ sur l'alphabet binaire $\{a,\ b\}$ par une expression régulière ou par un **AFD** (le langage n'est pas régulier).

L'analyse syntaxique est un peu plus compliquée que l'analyse lexicale et nécessite des méthodes plus avancées. Cependant, la même stratégie de base est utilisée : une **notation** très adaptée pour les programmeurs humains est convertie en une **machine automatique** (**reconnaisseur**) dont l'exécution est plus efficace.

Un programmeur humain manipule une notation plus facile à utiliser et à comprendre : les **grammaires hors-contexte** (**GHC**). En théorie des langages, une **GHC** est une notation **récursive** pour décrire certains langages. Les **GHC** trouvent une importante application dans les spécifications des langages de programmation. Ce sont des notations concises qui permettent de décrire la syntaxe des langages de programmation classiques. De plus, il est possible de **transformer mécaniquement** une **GHC** en un **analyseur syntaxique**. L'analyseur fait apparaître la structure du programme source, souvent sous la forme d'un arbre d'expression pour chaque instruction du programme.

5.1 Notion de GHC

Les **GHC** constituent un outil formel pour décrire quels sont les flux de lexèmes corrects et comment ils doivent être structurés. L'analyseur lexical décrit comment former les lexèmes du langage. L'analyseur syntaxique décrit ensuite comment

assembler les lexèmes pour former des phrases correctes. Une **GHC** fournit une description des phrases (*programmes*) syntaxiquement corrects.

Définition 5.1.1 *Une **GHC** est un 4-uplet $G = (V, T, P, S)$ où :*

- ▷ V *: un ensemble fini. C'est l'**alphabet des symboles variables** ou **non-terminaux**.*
- ▷ T *: un ensemble fini disjoint de V ($V \cap T = \emptyset$). C'est l'**alphabet des symboles terminaux**).*
- ▷ P *: ensemble fini d'éléments appelés **règles de production**.*
- ▷ $S \in V$ *appelé **axiome** (ou **symbole de départ**).*

Les **terminaux** sont en fait les lexèmes qui seront délivrés par l'analyseur lexical. Les **non-terminaux** sont des "**méta-symboles**" qui servent pour décrire les diverses constructions d'un langage (instructions, blocs, boucles, expressions, ...). Les **règles de production** sont des "**règles de réécriture**" qui montrent comment certains terminaux et non-terminaux se combinent pour former une contruction donnée. L'**axiome** est un "**méta-symbole particulier**", puisque ce dernier reflète la construction générale d'un langage (il identifie tout le programme). Par convention, les symboles **non-terminaux** seront notés en majuscule, alors que les symboles **terminaux** seront notés en minuscule.

Définition 5.1.2 *Dans une **GHC** $G = (V, T, P, S)$, une **forme** α sur G est un mot sur l'alphabet $(V \cup T)$, c'est-à-dire $\alpha \in (V \cup T)^*$.*

Définition 5.1.3 *Dans une **GHC** $G = (V, T, P, S)$, une **règle de production** est un couple $(X,\ \alpha)$ où $X \in V$ et α est une forme de G. On note une règle de production $(X,\ \alpha)$ par : $X \rightarrow \alpha$ qui se lit "X peut se réécrire comme α". Si $(X,\ \alpha)$ et (X, β) sont deux règles de productions telles que $\alpha \neq \beta$, on écrit : $X \rightarrow \alpha \mid \beta$.*

Exemple 5.1.1 *Soit $G = (V, T, P, S)$ avec :*

- $V = \{S, A, B\}$.
- $T = \{a, b\}$.

- $P = \{S \to AB \mid aS \mid A, A \to Ab \mid \varepsilon, B \to AS\}$.
- S *: l'axiome.*

Pour cette grammaire, les mots AB, aaS *et* ε *sont des formes sur* G.

5.2 Dérivations

Définition 5.2.1 *Soient* $G = (V, T, P, S)$ *une* ***GHC****,* β *et* δ *deux* ***formes*** *sur* G*. On dit que la forme* δ ***dérive en une seule étape*** *de la forme* β*, s'il existe une règle de production* $X \to \alpha$ *et deux formes* u *et* v *telles que* $\beta = uXv$ *et* $\delta = u\alpha v$*. On écrit dans ce cas :* $\beta \Rightarrow^1 \delta$ *ou, pour simplifier* $\beta \Rightarrow \delta$.

Exemple 5.2.1 *Dans la* ***GHC*** *de l'exemple* 4.3.1*, on a :* $AB \Rightarrow AAS$*. En effet, la production* $B \to AS$ *permet de substituer* B *par* AS *dans* AB*. Ce qui donne* AAS.

Définition 5.2.2 *Soient* $G = (V, T, P, S)$ *une* ***GHC****,* β *et* δ *deux* ***formes*** *sur* G *et* $k \geq 1$ *un entier. On dit que* δ ***dérive en k étapes*** *de* β*, s'il existe* $(k+1)$ *formes* $\beta_0,\ \beta_1,\ ...,\ \beta_k$ *telles que :*

1. $\beta_0 = \beta$.
2. $\beta_i \Rightarrow \beta_{i+1}, \forall i \in \{0, 1, ..., k-1\}$.
3. $\beta_k = \delta$

Dans ce cas, on écrit : $\beta \Rightarrow^k \delta$*. L'entier* k *s'appelle la* ***longueur de la dérivation****.*

Définition 5.2.3 *Soient* $G = (V, T, P, S)$ *une* ***GHC****,* β *et* δ *deux* ***formes*** *sur* G*. On dit que* δ ***dérive en plusieurs étapes*** *de* β*, s'il existe un entier* $k \geq 1$*, tel que* $\beta \Rightarrow^k \delta$*. On écrit pour simplifier :* $\beta \Rightarrow^* \delta$*. Par convention,* $\beta \Rightarrow^0 \beta$.

Exemple 5.2.2 *Soit la* ***GHC*** *de l'exemple* 4.3.1*. On a* $AB \Rightarrow^8 bba$*. En effet, on a :*

- $AB \Rightarrow AbB$ (*règle* $A \to Ab$).
- $AbB \Rightarrow AbbB$ (*règle* $A \to Ab$).
- $AbbB \Rightarrow bbB$ (*règle* $A \to \varepsilon$).
- $bbB \Rightarrow bbAS$ (*règle* $B \to AS$).

- $bbAS \Rightarrow bbS$ (***règle*** $A \rightarrow \varepsilon$).
- $bbS \Rightarrow bbaS$ (***règle*** $S \rightarrow aS$).
- $bbaS \Rightarrow bbaA$ (***règle*** $S \rightarrow A$).
- $bbaA \Rightarrow bba$ (***règle*** $A \rightarrow \varepsilon$).

Lemme 5.2.1 (***fondamental***) *Soient $G = (V, T, P, S)$ une **GHC** et α, α', β, β' et δ des **formes** sur G. Si $\alpha \Rightarrow^* \alpha'$ et $\beta \Rightarrow^* \beta'$, alors :*

1. $\alpha\beta \Rightarrow^* \alpha'\beta$.
2. $\alpha\beta \Rightarrow^* \alpha\beta'$.
3. $\alpha\beta \Rightarrow^* \alpha'\beta'$.

Inversement, si $\alpha\beta \Rightarrow^* \delta$, alors il existe deux formes α' et β' telles que :

1. $\delta = \alpha'\beta'$.
2. $\alpha \Rightarrow^* \alpha'$.
3. $\beta \Rightarrow^* \beta'$.

5.3 Langage engendré par une GHC

Définition 5.3.1 *Soit $G = (V, T, P, S)$ une **GHC**. Le **langage engendré** par G est formé par tous les mots sur l'alphabet T qui **dérivent en plusieurs étapes** de l'**axiome** S de G. On le note par $L(G)$. Formellement :*

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

Exemple 5.3.1 *Soit la **GHC** $G = (\{S\}, \{a\}, P, S)$ définie par les productions : $S \rightarrow aaS \mid \varepsilon$. Alors, $L(G) = \{(aa)^n \mid n \geq 0\} = (aa)^*$.*

Exemple 5.3.2 *Soit la **GHC** $G = (\{S\}, \{a, b\}, P, S)$ définie par les productions $S \rightarrow aSb \mid \varepsilon$. On a : $L(G) = \{a^n b^n \mid n \geq 0\}$.*

5.4 Langages hors contexte

5.4.1 Définition

Définition 5.4.1 *Soit L un langage sur un alphabet A. On dit que L est **hors contexte** (ou **algébrique**) sur A, si et seulement si L est engendré par une **GHC**, c'est-à-dire il existe une **GHC** $G = (V, T, P, S)$ tel que $A = T$ et $L(G) = L$.*

Exemple 5.4.1 *Les langages suivants sont algébriques :*

- $L = \{(aa)^n \mid n \geq 0\} = (aa)^*$. *En effet, il est engendré par la grammaire :* $S \to \varepsilon \mid aaS$.
- $L = \{a^n b^n \mid n \geq 0\}$, *car engendré par la grammaire :* $S \to \varepsilon \mid aSb$.

Théorème 5.4.1 *Tout langage régulier sur un alphabet A est algébrique sur A.*

Remarque 5.4.1 *La réciproque de ce théorème est fausse. Voici un contre exemple : le langage $L = \{a^n b^n \mid n \geq 0\}$ est algébrique, mais non régulier.*

5.4.2 Propriétés de clôture pour langages algébriques

Théorème 5.4.2 *La classe des langages algébriques est **close** par opérations **rationnelles** et par **substitution** :*

- ▷ ***Réunion** : si L et K sont algébriques, alors $L \cup K$ est algébrique.*
- ▷ ***Concaténation** : si L et K sont algébriques, alors LK est algébrique.*
- ▷ ***Étoile** : si L est algébrique, alors L^* est algébrique.*
- ▷ ***Substitution** : si L et K sont algébriques, alors le langage obtenu en substituant toutes les lettres a dans tous les mots $\omega \in L$ par K est algébrique.*

5.5 Propriétés de non clotûre

Théorème 5.5.1 *La classe des langages algébriques n'est pas close par les opérations d'intersection et de complémentation.*

Contre-exemple pour l'intersection. Soient les deux langages : $L = \{a^p b^p c^q \mid p, q \geq 0\}$ et $M = \{a^p b^q c^q \mid p, q \geq 0\}$. Les langages L et M sont algébriques, car $L = \{a^p b^p \mid p \geq 0\}\{c^q \mid q \geq 0\}$ et $M = \{a^p \mid p \geq 0\}\{b^q c^q \mid q \geq 0\}$, et l'on on sait que $\{a^p b^p \mid p \geq 0\}$, $\{b^q c^q \mid q \geq 0\}$, $\{a^p \mid p \geq 0\}$ et $\{c^q \mid q \geq 0\}$ sont algébrique. Cependant, le langage $L \cap M = \{a^p b^p c^p \mid p \geq 0\}$ n'est pas algébrique.

Preuve pour le complémentaire. Si la classe des langages algébriques est close par complémentaire, alors elle le serait par intersection. En effet, la classe des langages algébriques est close par réunion et l'on a toujours : $L \cap M = \overline{\overline{L} \cup \overline{M}}$. Or, la classe des langages algébriques n'est pas close par intersection.

5.6 Arbres de dérivation

Définition 5.6.1 *Soit $G = (V, T, P, S)$ une **GHC**. Un **arbre de dérivation** de G est un **arbre** Ω fini **étiquetté** par les éléments de $(V \cup T)$ et vérifiant les propriétés suivantes :*

- ▷ *Les **nœuds internes** de Ω sont les symboles non-terminaux $(X \in V)$.*
- ▷ *Si un **nœud interne** X de Ω, a pour fils $u_1,\ u_2,\ ...,\ u_n$, alors $X \rightarrow u_1u_2...u_n$ est une règle production de G.*

Définition 5.6.2 *Soient $G = (V, T, P, S)$ une **GHC** et Ω un arbre de dérivation de G. La **frontière** de Ω, notée $Fr(\Omega)$, est le mot obtenu par concaténation (de gauche à droite) des étiquettes de ses feuilles.*

Définition 5.6.3 *Soient $G = (V, T, P, S)$ une **GHC** et Ω un arbre de dérivation de G. On dit que Ω est **complet** (ou **total**) si :*

- *La **racine** de Ω est l'**axiome** S de G.*

1. *La **frontière** de Ω est un **mot terminal**, c'est-à-dire un mot du langage T^*.*

*Un arbre de dérivation est **incomplet** ou **partiel** s'il n'est pas complet.*

Exemple 5.6.1 *Soit G la **GHC** définie par : $S \rightarrow aB$ et $B \rightarrow bc|bB$. On a la dérivation : $S \Rightarrow aB \Rightarrow abc$. Un arbre de dérivation associé est le suivant :*

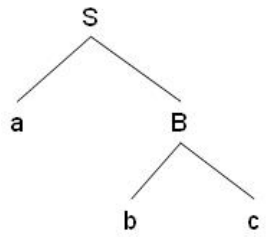

FIG. 5.1 – Un arbre de dérivation.

Proposition 5.6.1 *Soient $G = (V, T, P, S)$ une **GHC**. Un mot $\omega \in T^*$ appartient au langage $L(G)$ engendré par G, si et seulement si, il existe un arbre de dérivation total Ω de G ayant comme frontière le mot ω.*

Définition 5.6.4 *On appelle **dérivation la plus à gauche**, une dérivation où la **variable substituée** est systématiquement le symbole le **plus à gauche** du second membre de la règle de production.*

Exemple 5.6.2 *Soit G la **GHC** définie par : $S \rightarrow AB|AC$, $A \rightarrow x|xy$, $B \rightarrow z$ et $C \rightarrow yz$. Le mot $\omega = xz$ peut être obtenu par la dérivation la plus à gauche suivante : $S \Rightarrow AB \Rightarrow xB \Rightarrow xz$.*

Définition 5.6.5 *On appelle **dérivation la plus à droite**, une dérivation où la **variable substituée** est systématiquement le symbole le **plus à droite** du second membre de la règle de production.*

Exemple 5.6.3 *Soit G la **GHC** définie par : $S \rightarrow AB|AC$, $A \rightarrow x|xy$, $B \rightarrow z$ et $C \rightarrow yz$. Le mot $\omega = xz$ peut être obtenu par la dérivation la plus à droite suivante : $S \Rightarrow AB \Rightarrow Az \Rightarrow xz$.*

5.7 Ambiguïté

Définition 5.7.1 *Une **GHC** est dite **ambiguë**, s'il existe un mot $\omega \in L(G)$ ayant au moins deux arbres de dérivation distinctes.*

Définition 5.7.2 *Un langage L est dit **non ambigu**, s'il existe au moins une **GHC** **non ambiguë** qui l'engendre. Sinon, L est dit **inhéremment ambigu**.*

Exemple 5.7.1 *Soit G la **GHC** définie par : $S \rightarrow AB|AC$, $A \rightarrow x|xy$, $B \rightarrow z$ et $C \rightarrow yz$. La grammaire G est ambiguë. En effet, le mot $\omega = xyz$ possède deux arbres de dérivation distincts :*

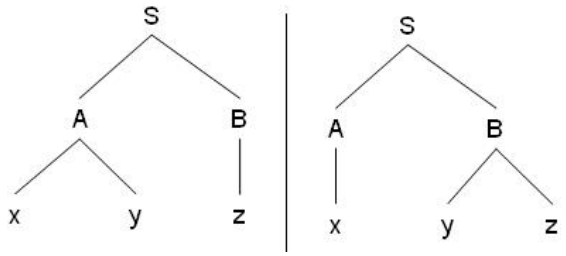

FIG. 5.2 – Deux arbres de dérivation distincts pour le mot $w = xyz$.

5.8 Réécriture de grammaires

5.8.1 Élimination de la récursivité immédiate à gauche

Définition 5.8.1 *Une **GHC** G est **immédiatement récursive à gauche** (**IRG**), si elle contient au moins une production de la forme $X \to X\alpha$, où α est une forme sur G.*

Exemple 5.8.1 *La grammaire : $S \to ScB \mid C$, $B \to Ba \mid \varepsilon$ et $C \to Cc \mid b \mid d$, est **IRG** (elle contient $S \to ScB$).*

Méthode d'élimination de la RIG :

Algorithm 6 Élimination de la RIG

1: **Transformer** toute production de la forme $X \to X\alpha$ par: $X \to \beta Y$ et $Y \to \alpha Y \mid \varepsilon$.

FIG. 5.3 – Algorithme de l'élimination de la **RIG**.

Exemple 5.8.2 *La grammaire précédente est équivalente à : $S \to CX$, $X \to cBX \mid \varepsilon$, $B \to Y$, $Y \to aY \mid \varepsilon$, $C \to bZ \mid dZ$ et $Z \to cZ \mid \varepsilon$.*

5.8.2 Élimination de la récursivité à gauche

Définition 5.8.2 *Une **GHC** G est **récursive à gauche**, si elle contient au moins un **non-terminal** X tel que $X \Rightarrow^+ X\alpha$, où α est une forme sur G.*

Exemple 5.8.3 *La grammaire : $S \to Ba \mid b$ et $B \to Bc \mid Sd \mid \varepsilon$ est récursive à gauche, puisque $S \Rightarrow Ba \Rightarrow Sda$.*

Algorithme de l'élimination de la RG :

Algorithm 7 Élimination de la RG

1: Ordonner les non-terminaux X_1, X_2, ..., X_n;
2: **for** $(i := 1; i \leq n; i++)$ **do**
3: **for** $(j := 1; i \leq i - 1; j++)$ **do**
4: **Remplacer** chaque production $X_i \rightarrow X_j\alpha$ par $X_i \rightarrow \beta_1\alpha \mid ... \mid \beta_k\alpha$, où $X_j \rightarrow \beta_1 \mid ... \mid \beta_k$ sont les X_j-productions courantes.
5: **Éliminer** les **RIG** des X_i-productions.
6: **end for**
7: **end for**

FIG. 5.4 – Algorithme de l'élimination de la **RG**.

Exemple 5.8.4 *Soit la grammaire :* $S \rightarrow Ba \mid b$ *et* $B \rightarrow Bc \mid Sd \mid \varepsilon$.

1. *On ordonne les non-terminaux :* S, B *(*$n = 2$*).*

2. *Itération* $i = 1$*. La boucle sur* j *ne s'exécute pas.*

3. *Itération* $i = 2$*. La boucle sur* j *s'exécute une seule fois (*$j = 1$*).*

   - *On remplace la production* $B \rightarrow Sd$ *par* $B \rightarrow Bad \mid bd$.
   - *On élimine les récursivités immédiates à gauche* $B \rightarrow Bc$ *et* $B \rightarrow Bad$. *On obtient,* $B \rightarrow bdX \mid X$ *et* $X \rightarrow cX \mid adX \mid \varepsilon$.

5.8.3 Factorisation à gauche

Définition 5.8.3 *Une* ***GHC*** G *est* ***non factorisée à gauche****, si elle contient au moins une* ***alternative*** *de la forme* $X \rightarrow \alpha\beta \mid \alpha\delta$ *où* α, β *et* δ *sont des formes sur* G.

Exemple 5.8.5 *La grammaire :* $S \rightarrow aEbS \mid aEbSeB \mid a$, $E \rightarrow bcB \mid bca$ *et* $B \rightarrow ba$ *n'est pas factorisée à gauche.*

Algorithme de factorisation à gauche :

Algorithm 8 Factorisation à gauche

1: **repeat**
2: **for** (X non-terminal) **do**
3: Trouver α, le **plus long préfixe commun** à 2 alternatives ou plus.
4: **if** ($\alpha \neq \varepsilon$) **then** ▷ il y a un préfixe commun non trivial
5: **Remplacer** toutes les X-productions par : $X \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_k \mid \delta$ où δ représente toutes les alternatives qui ne commencent pas par le préfixe α, par: $X \rightarrow \alpha Y \mid \delta$ et $Y \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_k$.
6: **end if**
7: **end for**
8: **until** (Aucune des alternatives d'un même non-terminal n'ait de préfixe commun)

FIG. 5.5 – Algorithme de factorisation à gauche.

Exemple 5.8.6 *Pour la grammaire du dernier exemple, on obtient :*

- *Itération* 1. *On a les productions :* $S \rightarrow aEbSX \mid a$, $X \rightarrow eB \mid \varepsilon$, $E \rightarrow bcY$, $Y \rightarrow B \mid a$, $B \rightarrow ba$.

- *Itération* 2. *On a les productions :* $S \rightarrow aZ$, $Z \rightarrow EbSX \mid \varepsilon$, $X \rightarrow eB \mid \varepsilon$, $E \rightarrow bcY$, $Y \rightarrow B \mid a$, $B \rightarrow ba$.

5.9 Précédence des opérateurs

Certains analyseurs nécessitent que la grammaire de base soit non ambiguë. La solution pour utiliser ce type d'analyseurs consiste alors à réécrire la grammaire initiale sous une autre forme qui évite toutes les sortes d'ambiguïté. Dans le cas d'une grammaire manipulant des opérateurs, il faut exprimer la hiérarchie des opérateurs dans la grammaire elle-même.

5.9.1 Associativité des opérateurs

Soit $\oplus$ un opérateur quelconque.

Définition 5.9.1 *On dit que* $\oplus$ *est :*

1. ***associatif à gauche**, si l'expression $a \oplus b \oplus c$ doit être évaluée de gauche à droite, c'est-à-dire comme $(a \oplus b) \oplus c$.*

2. *a une **associatif à droite**, si l'expression $a \oplus b \oplus c$ doit être évaluée de droite à gauche, c'est-à-dire comme $a \oplus (b \oplus c)$.*

3. *est **non associatif**, si les expressions de la forme $a \oplus b \oplus c$ sont illégales.*

Voici des exemples :

- Par la convention usuelle, les opérateurs $-$ et $/$ sont associatifs à gauche. Les opérateurs $+$ et $*$ sont associatifs au sens mathématique. Cela signifie que l'ordre d'évaluation d'une expression $a + b + c$, n'a pas d'intérêt. Cependant, pour interdire l'ambiguïté, il faut fixer une associativité pour ces opérateurs. Par convention, les opérateurs $+$ et $*$ sont associatifs à gauche.

- Les opérateurs de construction de liste dans les langages fonctionels (comme :: et @ en SML, sont typiquement associatifs à droite. L'opérateur d'affectation est également associatif à droite (l'expression $a = b = c$ est évaluée comme $a = (b = c)$).

- Dans certains langages (comme $Pascal$), les opérateurs de comparaison (comme $<$ ou $>$) sont non associatifs, ce qui explique que l'expression $2 < 3 < 4$ est syntaxiquement incorrecte en $Pascal$.

5.9.2 Élimination de l'ambiguïté des opérateurs

Soit la grammaire ambiguë suivante qui manipule un opérateur :

$$E \rightarrow E \oplus E$$

$$E \rightarrow \textbf{num}$$

On peut réécrire cette grammaire de manière à obtenir une grammaire équivalente mais non ambiguë. L'associativité de l'opérateur $\oplus$ influencera le choix des productions convenables :

- Si $\oplus$ n'est pas **associatif**, nous transformons la grammaire comme suit :

$$E \rightarrow E' \oplus E' \mid E'$$

$$E' \rightarrow \textbf{num}$$

Notons que cette transformation modifie le langage généré par la grammaire initiale, parce qu'elle rend illégales les expressions de la forme $num \oplus num \oplus num$.

- Si $\oplus$ est **associatif à gauche**, nous transformons la grammaire comme suit :

$$E \rightarrow E \oplus E' \mid E'$$

$$E' \rightarrow \textbf{num}$$

Maintenant, l'expression $2 \oplus 3 \oplus 4$ possède un seul arbre de dérivation :

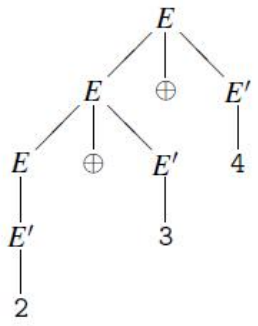

FIG. 5.6 – L'arbre de dérivation de l'expression $2 \oplus 3 \oplus 4$.

- Si $\oplus$ est **associatif à droite**, nous transformons la grammaire comme suit :

$$E \rightarrow E' \oplus E \mid E'$$

$$E' \rightarrow \textbf{num}$$

Avec cette transformation, l'expression $2 \oplus 3 \oplus 4$ possède maintenant un seul arbre de dérivation conforme avec l'associativité à droite de l'opérateur $\oplus$:
Pour que ces règles marchent, il faut que les opérateurs ayant la même précédence aient la même associativité. En effet, mélanger la récursivité à gauche et la récursivité à droite pour un même non-terminal rend la grammaire ambiguë. Considérons la grammaire défine par :

$$E \rightarrow E \ominus E'$$

$$E \rightarrow E' \oplus E$$

$$E \rightarrow E'$$

$$E' \rightarrow \textbf{num}$$

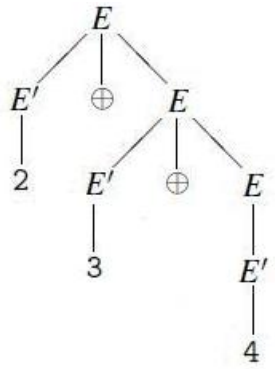

FIG. 5.7 – L'arbre de dérivation de l'expression $2 \oplus 3 \oplus 4$.

Cette grammaire donne aux opérateurs $\ominus$ et $\oplus$ la même précédence et des associativités différentes (à gauche pour $\ominus$ et à droite pour $\oplus$). Elle n'est pas seulement ambiguë, mais elle n'accèpte même pas le langage attendu. Par exemple, la chaîne **num** $\ominus$ **num** $\oplus$ **num** n'est pas dérivable par cette grammaire.

En général, il n'y a pas un moyen évident pour résoudre l'ambiguïté dans une expression comme $1 \ominus 2 \oplus 3$, où $\ominus$ est associatif à gauche et $\oplus$ est associatif à droite (et vice-versa). C'est pour cette raison que la plupart des langages de programmation (et la plupart des générateurs d'analyseurs syntaxiques) forcent cette règle :

Deux opérateurs ayant un même niveau de précédence (priorité) doivent avoir la même associativité.

Il est naturel de trouver des opérateurs de priorités différentes dans une même expression, comme $2 + 3 * 4$. La suppression d'ambiguïté dans les grammaires favorisant ce type d'expressions est garantie par l'utilisation d'un symbole non-terminal pour chaque niveau de priorité. L'idée est que si une expression utilise un opérateur d'un certain niveau de priorité n, alors ses sous-expressions ne peuvent pas utiliser des opérateurs de priorité inférieure à n, sauf si elles sont incluses dans des parenthèses. Par conséquent, les productions pour un non-terminal correspondant à un niveau de priorité particulier feront référence uniquement aux non-terminaux qui correspondent au même niveau de priorité ou à un niveau de priorité supérieur. L'exemple suivant montre comment ces considérations sont prises en compte pour avoir une grammaire d'expressions arithmétiques où les opérateurs $+$, $-$, $*$ et $/$ ont tous une associativité à gauche, $+$ et $-$ ont une même priorité qui est inférieure à celle de $*$ et $/$, en plus les parenthèses ont la plus grande priorité :

$Exp \rightarrow Exp + Exp1$
$Exp \rightarrow Exp - Exp1$
$Exp \rightarrow Exp1$
$Exp1 \rightarrow Exp1 * Exp2$
$Exp1 \rightarrow Exp1 / Exp2$
$Exp1 \rightarrow Exp2$
$Exp2 \rightarrow$ **num**
$Exp2 \rightarrow (Exp)$

Avec cette grammaire, l'expression $2+3*4$ n'a qu'une seule dérivation possible :

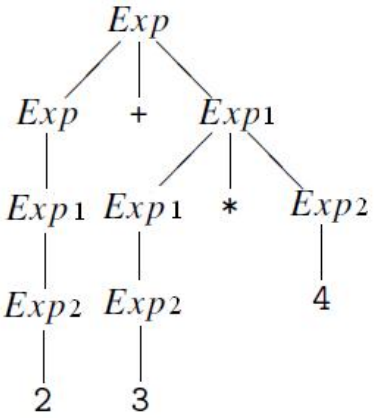

FIG. 5.8 – L'arbre de dérivation de l'expression $2 + 3 * 4$.

Chapitre 6

Analyse syntaxique

6.1 Rôle de l'analyse syntaxique

L'analyse syntaxique est assurée par un module appelé **analyseur syntaxique** (*parser*). Le rôle de l'analyseur syntaxique est de combiner les unités lexicales fournits par l'analyseur lexical pour former des **structures grammaticales**. Ces dernières sont typiquement de la forme d'une structure de données d'arbre appellée **arbre syntaxique abstrait** (*Abstract Syntax Tree*, ou **AST**). Les **feuilles** de cet arbre correspondent aux lexèmes délivrés par l'analyseur lexical. La séquence formée par ces feuilles lorsuq'elles sont lues de gauche à droite, est identique au texte d'entrée initial. Une autre tâche de l'analyseur syntaxique consiste à **détecter** les erreurs syntaxiques dans le texte source. Par exemple :

- *Oubli d'un point virgule* à la fin d'une instruction :

  ```
  int a
  ```

- *Manque d'une parenthèse droite* :

  ```
  if(a == 0 printf("zero\n");
  ```

- *Opérande attendu* dans une opération :

  ```
  a = b * ;
  ```

La figure suivante illustre la position de l'analyseur syntaxique dans le front-end

d'un compilateur :

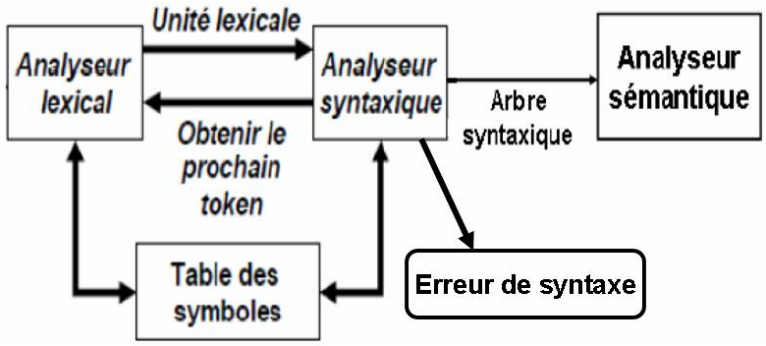

FIG. 6.1 – Position de l'analyseur syntaxique.

6.2 Analyse syntaxique et GHC

L'analyseur syntaxique doit vérifier la validité grammaticale des "phrases" du programme source et déterminer leur structure sous forme d'arbres de syntaxe.

Le problème de l'analyse syntaxique peut se formuler comme suit : étant données une grammaire hors-contexte $G = (V,\ T,\ P,\ S)$ et un terminal $\omega \in T^*$, un analyseur syntaxique doit répondre à la question "est-ce que $\omega \in T^*$? Si la réponse est affirmativen il doit construire l'arbre de syntaxe de ω. Sinon, il doit signaler une erreur de syntaxe.

On peut distinguer les méthodes d'analyse syntaxique par l'ordre dans lequel elles construisent les nœuds de l'arbre syntaxique. Il y a deux classes de méthodes d'analyse syntaxique :

1. Les **méthodes d'analyse descendante** : elles construisent l'arbre syntaxique en **préordre**, c'est-à-dire du haut en bas, en partant de la racine (l'axiome) vers les feuilles (les lexèmes).

2. Les **méthodes d'analyse ascendante** : elles construisent l'arbre syntaxique en **postrdre**, c'est-à-dire du bas en haut, en partant des feuilles vers la racine.

6.3 Analyse descendante

6.3.1 Principe

Un **analyseur descendant** construit l'arbre de dérivation de la chaîne d'entrée, en partant de la **racine** (l'axiome de la grammaire) et en créant les nœuds de l'arbre en **préordre** jusqu'à les feuilles (la chaîne de terminaux).

6.3.2 Un premier exemple

Considérons la grammaire définie par :

$$S \rightarrow aSbT \mid cT \mid d$$

$$T \rightarrow aT \mid bS \mid c$$

Faisons l'analyse du mot $w = accbbadbc$. On part avec l'arbre contenant le seul sommet S. On lit le premier lexème du mot : a. Il y a une seule S-production dont le corps commence par a : $S \rightarrow aSbT$. On applique cette production pour développer l'arbre de dérivation, ce qui donne :

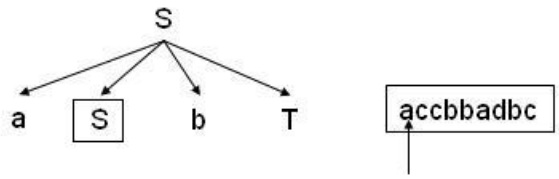

FIG. 6.2 – Première dérivation du mot $w = accbbadbc$.

Nous avançons le pointeur de lecture vers le lexème suivant (à gauche) : c. Cherchons à faire une dérivation **la plus à gauche**. Le prochain symbole non-terminal à développer est S. La production à appliquer est $S \rightarrow cT$, ce qui donne :

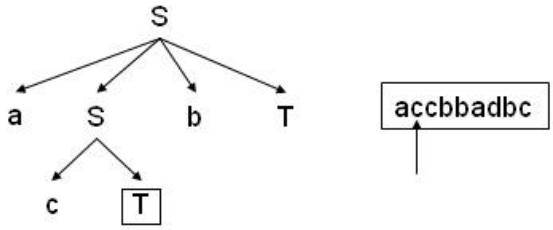

FIG. 6.3 – Deuxième dérivation du mot $w = accbbadbc$.

Nous répétons ce processus jusqu'à obtenir l'arbre de dérivation totale (voir figure 6.4). Sur cet exemple, l'analyse est très simple car chaque production commence par un terminal différent, et par conséquent le choix de la prochaîne production à appliquer ne pose aucun problème.

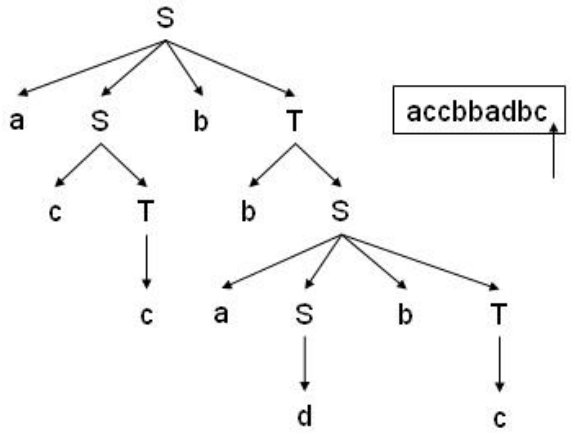

FIG. 6.4 – Arbre de dérivation totale du mot $w = accbbadbc$.

6.3.3 Un deuxième exemple

Considérons maintenant la grammaire définie par :

$$S \rightarrow aAb$$
$$A \rightarrow cd \mid c$$

Faisons l'analyse du mot $w = acb$. On part de la racine S. On lit le premier lexème du mot : a. Il y a une seule production à appliquer : $S \rightarrow aAb$ et le prochain non-terminal à développer est A, ce qui donne :

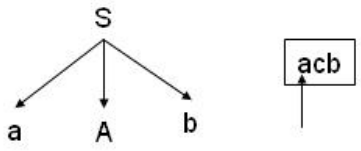

FIG. 6.5 – Première dérivation du mot $w = acb$.

En lisant le prochaîn lexème c, on ne sait pas s'il faut prendre la production $A \rightarrow cd$ ou la production $A \rightarrow c$. Pour le savoir, il faut lire aussi le lexème suivant b, ou

alors, il faut donner la possibilité de faire des **retours en arrière** (*backtracking*) : on choisit la production $A \rightarrow cd$, on aboutit à un échec, ensuite on retourne en arrière et on choisit la production $A \rightarrow c$ qui convient cette fois-ci.

6.4 Analyse prédictive

L'analyse syntaxique **prédictive** est une méthode d'analyse descendante dans laquelle nous pouvons toujours **choisir une production unique** en se basant sur le **prochain symbole de l'entrée** et sans effectuer aucun **retour en arrière**.

Il y a deux façons pour effectuer une analyse prédictive :

- ▷ L'**analyse prédictive récursive** : implémentée en utilisant des **procédures récursives**.
- ▷ L'**analyse prédictive itérative** : dirigée par une **table d'analyse**.

L'analyse prédictive nécessite le calcul de certaines fonctions : **Nullable**, **Premier** et **Suivant**.

6.4.1 Le prédicat Nullable

Définition 6.4.1 *Une forme α d'une grammaire est dite **nullable**, et l'on écrit $Nullable(\alpha) = Vrai$, si et seulement si l'on peut dériver le mot vide ε à partir de α (c'est-à-dire $\alpha \Rightarrow^* \varepsilon$).*

Les règles suivantes permettent de calculer le prédicat $Nullable$ d'une manière récursive :

1. $Nullable(\varepsilon) = Vrai$.
2. $Nullable(a) = Faux, \forall a \in T$.
3. $Nullable(\alpha\beta) = Nullable(\alpha) \wedge Nullable(\beta), \forall \alpha, \beta \in (V \cup T)^*$.
4. $Nullable(X) = Nullable(\alpha_1) \vee ... \vee Nullable(\alpha_n), \forall X \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$.

Définition 6.4.2 *Pour une grammaire G, on définit **Null**(G) comme étant le sous-ensemble des symboles **non-terminaux nullables**. Formellement :*

$$Null(G) = \{X \in V \mid Nullable(X) = Vrai\} = \{X \in V \mid X \Rightarrow^* \varepsilon\}$$

Exemple 6.4.1 *Soit la grammaire G définie par : $T \rightarrow R \mid aTc$ et $R \rightarrow bR \mid \varepsilon$*
On a le système suivant d'équations booléennes :

- $Nullable(T) = Nullable(R) \vee Nullable(aTc)$.
- $Nullable(R) = Nullable(\varepsilon) \vee Nullable(bR)$.
- $Nullable(aTc) = Nullable(a) \wedge Nullable(T) \wedge Nullable(c)$.
- $Nullable(bR) = Nullable(b) \wedge Nullable(R)$.
- $Nullable(\varepsilon) = Vrai$.

*La résolution de ce système montre que les non-terminaux T et R sont nullables : $Nullable(T) = Nullable(R) = Vrai$. Par suite, **Null**$(G) = \{T,\ R\}$.*

6.4.2 La fonction Premier (First)

Définition 6.4.3 *Soit $\alpha \in (V \cup T)^*$ une forme d'une **GHC** $G = (V,\ T,\ P,\ S)$. On définit l'ensemble **Premier**(α) (ou **First**(α)), comme étant l'ensemble des **terminaux** qui peuvent apparaître au début d'une forme dérivable à partir de α. D'une manière formelle :*

$$Premier(\alpha) = \{a \in T \mid \alpha \Rightarrow^* a\beta,\ \beta \in (V \cup T)^*\}$$

Les règles suivantes permettent de calculer cette fonction :

1. $Premier(\varepsilon) = \emptyset$.
2. $Premier(a) = \{a\}, \forall a \in T$.
3. Si $Nullable(\alpha) = Vrai$, $Premier(\alpha\beta) = Premier(\alpha)\ \cup\ Premier(\beta)$.
4. Si $Nullable(\alpha) = Faux$, $Premier(\alpha\beta) = Premier(\alpha)$.
5. $Premier(X) = Premier(\alpha_1) \cup ... \cup Premier(\alpha_n), \forall X \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$.

Exemple 6.4.2 *Soit la grammaire $T \rightarrow R \mid aTc$ et $R \rightarrow bR \mid \varepsilon$*
En appliquant les règles précédentes, on obtient le système suivant d'équations ensemblistes :

- $Premier(T) = Premier(R)\ \cup\ Premier(aTc)$.
- $Premier(R) = Premier(bR)\ \cup\ Premier(\varepsilon)$.

- $Premier(aTc) = Premier(a)$, *car* $Nullable(a) = Faux$.
- $Premier(bR) = Premier(b)$, *car* $Nullable(b) = Faux$.
- $Premier(\varepsilon) = \emptyset$.

On en déduit alors : $Premier(R) = \{b\}$ *et* $Premier(T) = \{a,\ b\}$.

6.4.3 La fonction Suivant (Follow)

Définition 6.4.4 *On définit l'ensemble* ***Suivant***(X) *(ou* ***Follow***(X)*) pour tout symbole non-terminal* $X \in V$*, comme étant l'ensemble des* ***terminaux*** *qui peuvent apparaître* ***juste après le symbole*** X *dans une dérivation à partir de l'****axiome*** S *de la grammaire* G*. Formellement :*

$$Suivant(X) = \{a \in T \mid S \Rightarrow^* \alpha X a \beta \; avec \; \alpha,\ \beta \in (V \cup T)^*\}$$

Pour traiter correctement les conditions de fin de chaîne d'entrée, nous avons besoin de détecter si $S \Rightarrow^* \alpha N$, c'est-à-dire s'il existe des dérivations dans lesquelles le symbole N peut être suivi immédiatement par la fin de l'entrée. Pour faciliter ce traitement, nous ajoutons une production supplémentaire à la grammaire initiale :

$$S' \rightarrow S\$$$

où S' est un nouveau symbole non-terminal qui remplace l'axiome S, et '$' un nouveau symbole terminal qui représente la fin de l'entrée. La nouvelle grammaire est dite la **grammaire augmentée** de G. Nous savons que :

$$\$ \in Suivant(N) \Leftrightarrow S' \Rightarrow^* \alpha N\$$$

ce qui est exactement le cas si $S \Rightarrow^* \alpha N$.

Lemme 6.4.1 *Soit la production* $M \rightarrow \alpha N \beta$ *d'une* ***GHC*** G*, où* M *et* N *sont deux symboles non-terminaux. On a :*

1. $Premier(\beta) \subseteq Suivant(N)$.
2. *Si* $Nullable(\beta) = Vrai$*, alors* $Suivant(M) \subseteq Suivant(N)$.

La première propriété est évidente, puisque β vient juste après le symbole non-terminal N. La seconde propriété résulte du fait que si $a \in Suivant(M)$, alors

par définition, il existe une dérivation de la forme $S \Rightarrow^* \gamma M a \delta$. Mais, comme $M \rightarrow \alpha N \beta$ et β est nullable, alors nous pouvons écrite :

$$S \Rightarrow^* \gamma M a \delta \Rightarrow^* \gamma \alpha N \beta a \delta \Rightarrow^* \gamma \alpha N a \delta$$

ce qui prouve que $a \in Suivant(N)$.

Si le corps d'une production contient plusieurs occurrences de non-terminaux, nous ajoutons des **contraintes** pour toutes ces occurrences, en fractionnant la partie droite de la production en les différents α, N et β.

Exemple 6.4.3 *Soit la production $A \rightarrow BcBdaAadC$. Voici les 4 fractionnements possibles :*

1. *Premier B : $\alpha = \varepsilon$, $N = B$ et $\beta = cBdaAadC$, qui génère la contrainte $\{c\} \subseteq Suivant(B)$.*
2. *Deuxième B : $\alpha = Bc$, $N = B$ et $\beta = daAadC$, qui génère la contrainte $\{d\} \subseteq Suivant(B)$.*
3. *Premier A : $\alpha = BcBda$, $N = A$ et $\beta = adC$, qui génère la contrainte $\{a\} \subseteq Suivant(A)$.*
4. *Premier C : $\alpha = BcBdaAad$, $N = C$ et $\beta = \varepsilon$, qui génère la contrainte $Suivant(A) \subseteq Suivant(C)$.*

Pour calculer la fonction **Suivant**(X) pour tous les non-terminaux X, il suffit alors de résoudre le système de contraintes obtenues en appliquant le lemme $4.14.1$, sans oublier d'ajouter la production $S' \rightarrow S\$$ de la grammaire augmentée.

Exemple 6.4.4 *Soit la grammaire définie par : $T \rightarrow R \mid aTc$ et $R \rightarrow RbR \mid \varepsilon$*
On ajoute la production $T' \rightarrow T\$$. On a alors :

1. *La règle $T' \rightarrow T\$$ ajoute la contrainte : $\{\$\} \subseteq Suivant(T)$.*
2. *La règle $T \rightarrow R$ ajoute la contrainte : $Suivant(T) \subseteq Suivant(R)$.*
3. *La règle $T \rightarrow aTc$ ajoute la contrainte : $\{c\} \subseteq Suivant(T)$.*
4. *La règle $R \rightarrow RbR$ ajoute les deux contraintes : $Suivant(R) \subseteq Suivant(R)$ qui est inutile, et $\{b\} \subseteq Suivant(R)$.*
5. *La règle $R \rightarrow \varepsilon$ n'ajoute rien.*

Ce qui donne finalement, $Suivant(T) = \{\$,\ c\}$ et $Suivant(R) = \{\$,\ c,\ b\}$.

6.5 Analyse LL(1)

Soient $X \in V$ le symbole **non-terminal courant à développer** et a le symbole **terminal de prévision** (le lexème suivant de l'entrée). Il est clair que si l'une de deux règles suivantes est vérifiée, alors on peut choisir la production $X \to \alpha$:

- $(R1)$: $a \in$ **Premier**(α).
- $(R2)$: Si **Nullable**$(\alpha) = Vrai$ et $a \in$ **Suivant**(X).

Si on peut toujours choisir une production uniquement en utilisant ces deux règles, l'analyse est dite $LL(1)$: c'est un cas particulier d'analyse prédictive. La première lettre L signifie "**Left-to-right scanning**", c'est-à-dire que l'analyseur lit la chaîne de l'entrée de gauche à droite. La seconde lettre L signifie "**Left-most derivation**", c'est-à-dire que l'analyseur effectue la dérivation la plus à gauche. Le nombre 1 indique qu'il y a **un seul symbole de prévision**. Si l'on utilise k symboles de prévision, l'analyse dans ce cas est appellée $LL(k)$.

6.5.1 Grammaires LL(1)

Définition 6.5.1 *Une grammaire est dite $LL(1)$, s'il est possible de l'analyser en utilisant une grammaire $LL(1)$.*

Toutes les grammaires ne sont pas des grammaires $LL(1)$. Par exemple, une grammaire **non factorisée à gauche** n'est pas une grammaire $LL(1)$. Par exemple, si X est le symbole non-terminal à développer et si a est le symbole de prévision et si la grammaire contient les deux productions $X \to aA \mid aB$, alors le choix d'une X-production est non déterministe. Nous admettons le théorème suivant :

Théorème 6.5.1 *Si une grammaire est $LL(1)$, alors elle est **non ambiguë**, **non récursive à gauche** et **factorisée à gauche**.*

La réciproque de ce théorème est fausse : il existe des grammaires non ambiguës, non récursives à gauche et factorisées à gauche, mais qui ne sont pas $LL(1)$. Le théorème suivant (admis) donne une caractérisation d'une grammaire $LL(1)$:

Théorème 6.5.2 *Une grammaire G est $LL(1)$, si et seulement si, chaque fois que $X \to \alpha \mid \beta$ sont deux X-productions distinctes de G, les conditions suivantes s'appliquent :*

1. ***Premier***$(\alpha) \cap$ ***Premier***$(\beta) = \emptyset$ *: pour tout terminal a, α et β ne se dérivent toutes les deux en des chaînes commençant par a.*

2. ***Nullable***$(\alpha) = Vrai \Leftrightarrow$ ***Nullable***$(\beta) = Faux$ *: les deux formes* α *et* β *ne peuvent être nullables toutes les deux à la fois.*
3. *Si* ***Nullable***$(\beta) = Vrai$*, alors* α *ne se dérive pas en une chaîne commençant par un terminal* $a \in$ `Suivant`(X).

Exemple 6.5.1 *La grammaire :* $T \rightarrow R \mid aTc$ *et* $R \rightarrow bR \mid \varepsilon$ *est* $LL(1)$.

Exemple 6.5.2 *La grammaire :* $T \rightarrow R \mid aTc$ *et* $R \rightarrow RbR \mid \varepsilon$ *n'est pas* $LL(1)$ *(violation de la troisième règle, car* $RbR \Rightarrow bR$ *et* bR *commence par le terminal* $b \in$ ***Suivant***(R)*).*

6.5.2 Implémentation d'un analyseur LL(1)

Un analyseur syntaxique $LL(1)$ s'implémente en utilisant l'une des deux approches suivantes :

- ▷ La **descente récursive** : la structure de la grammaire est directement traduite en une structure d'un programme d'analyse.
- ▷ L'utilisation d'une **table d'analyse** : cette dernière va guider le processus de choix des productions.

6.6 Analyseur syntaxique LL(1) par descente récursive

6.6.1 Principe

Comme son nom l'indique, la **descente récursive** est une technique d'analyse qui utilise des **procédures récursives** pour implémenter l'analsye syntaxique prédictive. L'idée centrale est que chaque symbole **non-terminal** dans la grammaire est **implémenté par une procédure** dans le programme d'analyse. Chaque telle procédure examine le **symbole suivant de l'entrée** afin de choisir une production. La partie droite de la production est alors analysée de la manière suivante :

- ▷ Un symbole **terminal** est mis en correspondance avec le **symbole suivant de l'entrée**. S'ils **concordent**, on se **déplace** sur le symbole suivant de l'entrée. Sinon, une **erreur syntaxique est rapportée**.
- ▷ Un symbole **non-terminal** est analysé en invoquant sa **procédure associée**.

6.6.2 Un exemple

Considérons la grammaire $\texttt{LL}(1)$ définie par :

$$T \rightarrow R \mid aTc$$
$$R \rightarrow bR \mid \varepsilon$$

On ajoute la règle de la grammaire augmentée : $T' \rightarrow T\$$. L'analyse syntaxique par descente récursive de cette grammaire sera effectuée à travers 3 procédures récursives : $T'()$, $T()$ et $R()$. Chaque procédure d'analyse $X()$ permet de simuler le choix d'une X-production. Pour l'analyse, on a besoin des éléments suivants :

- ▷ Une **variable globale** nommée `suivant` qui joue le rôle du **symbole de prévision**.
- ▷ Une **fonction globale** nommée **exiger**(*Lexème*), qui prend comme argument un symbole de l'entrée (un lexème) et le compare avec le prochain symbole de l'entrée (le symbole de prévision). S'il y a concordance, le prochain symbole est lu dans la variable `suivant`.
- ▷ La procédure **erreur**() qui reporte une erreur de syntaxe.

Voici le pseudo-code de cette fonction :

Algorithm 9 Fonction exiger($Lexeme\ token$)

Require: $token$: $Lexeme$;
1: **if** ($token ==$ suivant) **then**
2: $suivant :=$ `getLexeme()`;
3: **else**
4: `Erreur()`;
5: **end if**

FIG. 6.6 – Fonction exiger.

▷ **La procédure** $T'()$: le symbole non-terminal T' possède une seule production $T' \rightarrow T\$$, et donc le choix est trivial. Cependant, on doit ajouter une vérification sur le prochain symbole de l'entrée (la variable `suivant`). Si `suivant` est dans $\texttt{Premier}(T') = \{a,\ b,\ \$\}$, alors on choisira la production $T' \rightarrow T\$$. Cela s'exprime par l'appel de la procédure d'analayse associée au symbole T, suivi de la vérification du fin de l'entrée. Sinon, une erreur de syntaxe doit être raportée.

Algorithm 10 Procédure $T'()$

```
1: if (suivant ∈ {b, a, $}) then
2:     T();
3:     exiger('$');
4: else
5:     erreur();
6: end if
```

FIG. 6.7 – Procédure associée au symbole T'.

▷ **La procédure** $T()$: il y a deux productions issues de T : $T \rightarrow aTc$ et $T \rightarrow R$. Comme `Nullable`$(R) = Vrai$, alors on choisira la production $T \rightarrow R$, si `suivant` $\in Suivant(T)$ ou `suivant` $\in Premier(R)$, c'est-à-dire si `suivant` $\in \{a, b, \$\}$. Le choix de cette production se traduit alors par l'appel de la procédure d'analyse associée à R. La production $T \rightarrow aTc$ sera choisie sur le symbole de l'entrée $a \in$ `Premier`(aTc). Cela se traduit par l'appel de la procédure d'analyse associée à T, mais on doit vérifier le préfixe a et le suffixe c. Dans les autres cas, une erreur de syntaxe doit être rapportée.

Algorithm 11 Procédure $T()$

```
1: if (suivant ∈ {b, c, $}) then
2:     R();
3: else
4:     if (suivant == a) then
5:         exiger('a');
6:         T();
7:         exiger('c');
8:     else
9:         erreur();
10:    end if
11: end if
```

FIG. 6.8 – Procédure associée au symbole T.

▷ **La procédure** $R()$: il y a deux R-productions : $R \rightarrow bR$ et $R \rightarrow \varepsilon$. Comme `Nullable`$(\varepsilon) = Vrai$, alors on choisira la production $R \rightarrow \varepsilon$, si `suivant` $\in Suivant(R) = \{c, \$\}$. Le choix de cette production se traduit par l'instruction vide (Ne rien faire). La production $R \rightarrow bR$ sera choisie sur le symbole de l'entrée $b \in$ `Premier`(bR). Cela se traduit par l'appel de la procédure d'analyse associée

à R, avant lequel on doit vérifier le préfixe b. Dans les autres cas, une erreur de syntaxe doit être rapportée.

Algorithm 12 Procédure $R()$

```
1: if (suivant ∈ {c, $}) then
2:     Ne Rien Faire;
3: else
4:     if (suivant == b) then
5:         exiger('b');
6:         R();
7:     else
8:         erreur();
9:     end if
10: end if
```

FIG. 6.9 – Procédure associée au symbole R.

▷ Une fonction $parse()$ implémentera l'analyseur syntaxique. Au départ, on initialise la variable `suivant` au premier symbole de l'entrée (le premier lexème fournit) avant de lancer la procédure d'analyse globale $T'()$, donc il faut réaliser une première lecture (on alors effectue un appel à l'analyseur lexical). Ensuite, la procédure d'analyse $T'()$ est appelée. Celle-ci va appeler les autres et ainsi de suite. S'il n'y a aucune erreur de syntaxe, la procédure finira par s'exécuter sans problème.

Algorithm 13 Procédure $parse()$

```
1: suivant = getLexeme();
2: T'();
```

FIG. 6.10 – Procédure d'analyse syntaxique.

6.7 Analyseur syntaxique LL(1) dirigé par une table

6.7.1 Principe

Dans un analyseur $LL(1)$ dirigé par table, on encode la sélection des productions dans une table, appelée **table d'analyse** $LL(1)$, au lieu d'utiliser un programme. Ensuite, un programme (non récursif) utilisera cette table et une pile

pour effectuer l'analyse syntaxique.

Une table d'analyse $LL(1)$ est un tableau M à deux dimensions (une matrice) :

- ▷ Les lignes sont indexées par les symboles **non-terminaux**.
- ▷ Les colonnes sont indexées par les symboles **terminaux**, en plus du symbole de fin de l'entrée $\$$.
- ▷ Pour un symbole non-terminal X et un symbole terminal a, l'entrée $M[X, a]$ contient la **règle de production** qui sera appliquée lorsqu'on veut développer X sachant que le prochain symbole de l'entrée est a.

Cette décision est faite exactement comme dans le cas de l'analyse syntaxique $LL(1)$ par descente récursive :

- ▷ La production $X \rightarrow \alpha$ est dans $M[X, a]$, si et seulement si :
  - $a \in$ **Premier**(X).
  - $a \in$ **Suivant**(X) si **Nullable**$(\alpha) = Vrai$.

L'algorithme présenté dans la figure 6.11 permet de construire la table d'analyse $LL(1)$ d'une grammaire G.

Exemple 6.7.1 *Soit la grammaire définie par :*

$$T \rightarrow R \mid aTc$$
$$R \rightarrow bR \mid \varepsilon$$

Le tableau suivant montre la table d'analyse relative à cette grammaire :

	a	b	c	$\$$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow aTc$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow bR$	$R \rightarrow \varepsilon$	$R \rightarrow \varepsilon$

Proposition 6.7.1 *Une grammaire G est $LL(1)$, si et seulement si la table M d'analyse $LL(1)$ relative à G ne possède que des **entrées simples**, c'est-à-dire que $\forall X \in V$, $\forall a \in T \cup \{\$\}$: l'entrée $M[X, a]$ contient une **seule règle de production** ou bien elle est **vide**.*

Algorithm 14 Procédure de construction de la table $LL(1)$

```
1: for (X → α dans P) do
2:     for (a ∈ Premier(α)) do
3:         Ajouter X → α à M[X, a];
4:     end for
5:     if (Nullable(α) = Vrai) then
6:         for (b ∈ Suivant(X)) do
7:             Ajouter X → α à M[X, b];
8:         end for
9:         if ($ ∈ Suivant(X)) then
10:            Ajouter X → α à M[X, $];
11:        end if
12:    end if
13: end for
```

FIG. 6.11 – Algorithme de remplissage de la table d'analyse $LL(1)$.

La présence de deux ou plusieurs productions dans une entrée de la table d'analyse $LL(1)$ montre que la grammaire n'est pas $LL(1)$. Pour une grammaire $LL(1)$, une **entrée vide** dans la table d'analyse $LL(1)$ **indique la présence d'une erreur syntaxique**.

6.7.2 ALgorithme d'analyse syntaxique LL(1) dirigée par table

Comme le montre la figure, un analyseur syntaxique prédictif dirigé par table est composé de 4 éléments :

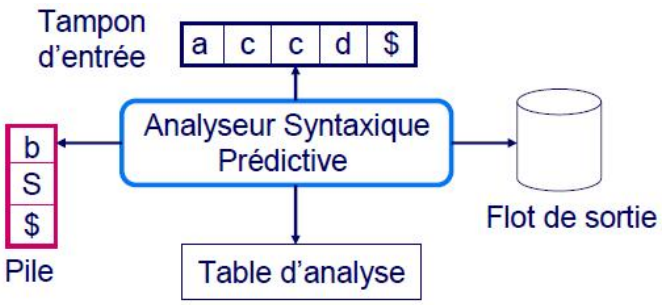

FIG. 6.12 – Les éléments pour un analyseur $LL(1)$ dirigé par table.

▷ Un **tampon d'entrée** : contient la chaîne à analyser, suivie du symbole $\$$.

- ▷ Une **pile** : contient une séquence de symboles (terminaux ou non-terminaux), avec un symbole $ qui marque le **fond de la pile**. Au départ, la pile contient l'**axiome** de la grammaire au sommet et le symbole $ au **fond**.

- ▷ Une **table d'analyse** $LL(1)$.

- Un **flot de sortie**.

Il y a trois cas possibles :

1. Si $X = a = \$$, l'analyseur s'arrête et annonce la réussite finale de l'analyse.
2. Si $X = a \neq \$$, l'analyseur dépile X et avance son pointeur d'entrée sur le symbole suivant (à droite).
3. Si X est un non-terminal, le programme consulte l'entrée $M[X, a]$ de la table d'analyse M. Cette entrée est soit une X-production, soit une erreur (entrée vide). Si par exemple, $M[X, a] = X \rightarrow \alpha_1\alpha_2...\alpha_n$, l'analyseur dépile X et empile, dans l'ordre $\alpha_n,\ \alpha_{n-1},\ ...,\ \alpha_1$. On a supposé, pour simplifier, que l'analyseur se contente pour tout résultat, d'imprimer la production utilisée. Cependant n'importe quelle autre action pourrait être exécutée à la place. Si $M[X, a] = ERREUR$ (cas d'une entrée vide), l'analyseur appelle une procédure de récupération sur erreur.

On peut visualiser l'analyse prédictive dirigée par table par un tableau à 3 colonnes :

- ▷ La première colonne contient la **configuration de la pile**.

- ▷ La deuxième colonne contient la **configuration de la chaîne de l'entrée**.

- ▷ La troisième colonne contient la **sortie** qui est tout simplement la règle de production appliquée.

Exemple 6.7.2 *Le tableau suivant illustre l'analyse prédictive $LL(1)$ dirigée par table de la grammaire définie par :*

$$T \rightarrow R \mid aTc$$

$$R \rightarrow bR \mid \varepsilon$$

pour la chaîne d'entrée $\omega = aabbbcc$:

$Etat\ de\ la\ pile$	$Entree$	$Sortie$
$\$T$	$aabbbcc\$$	$T \rightarrow aTc$
$\$cTa$	$aabbbcc\$$	
$\$cT$	$abbbcc\$$	$T \rightarrow aTc$
$\$ccTa$	$abbbcc\$$	
$\$ccT$	$bbbcc\$$	$T \rightarrow R$
$\$ccR$	$bbbcc\$$	$R \rightarrow bR$
$\$ccRb$	$bbbcc\$$	
$\$ccR$	$bbcc\$$	$R \rightarrow bR$
$\$ccRb$	$bbcc\$$	
$\$ccR$	$bcc\$$	$R \rightarrow bR$
$\$ccRb$	$bcc\$$	
$\$ccR$	$cc\$$	$R \rightarrow \varepsilon$
$\$cc$	$cc\$$	
$\$c$	$c\$$	
$\$$	$\$$	$ACCEPT$

6.8 Préparation d'une grammaire pour une analyse LL(1)

Pour développer un analyseur $LL(1)$ pour une grammaire, nous devons suivre les étapes suivantes :

1. Supprimer l'ambiguïté.
2. Éliminer la récursivité à gauche.
3. Factoriser à gauche.
4. Ajouter la règle de grammaire augmentée.
5. Calculer le prédicat $Nullable$ et la fonction $Premier$ pour chaque production, et la fonction $Suivant$ pour chaque symbole non-terminal.
6. Pour un non-terminal X à developper et un symble d'entrée a, choisir la production $X \rightarrow \alpha$ lorsque $a \in Premier(\alpha)$ et/ou si $Nullable(\alpha) = Vrai$ et $a \in Suivant(X)$.
7. Implémenter l'analyseur soit par un programme effectuant une descente récursive, soit par un programme itératif utilisant une table d'analyse $LL(1)$.

L'inconvénient majeur de l'analyse $LL(1)$ est que la majorité des grammaires nécessitent des réécritures supplémentaires. Même si ces réécritures peuvent être

effectuées d'une manière automatique, il reste un nombre important de grammaires qui ne peuvent pas être transformées automatiquement en grammaires $LL(1)$. Pour analyser de telles grammaires, nous devons recourir à de nouvelles techniques plus avancées.

Chapitre 7

L'outil Bison

7.1 Présentation de Bison

Bison est un constructeur d'analyseurs syntaxiques ascendents de type **LALR**. À partir d'une spécification d'une grammaire hors-contexteen, **Bison** génère le code C d'un analyseur syntaxique. Un fichier de spécification **Bison** est un fichier texte portant l'extension "$.y$". La compilation de ce fichier s'effectue par la commande "**bison**" et produit un fichier de même nom, mais avec une extension "$.tab.c$". Ce dernier contient le code C de l'analyseur. On obtient l'exécutable de l'analyseur en compilant ce dernier fichier via un compilateur C (gcc).

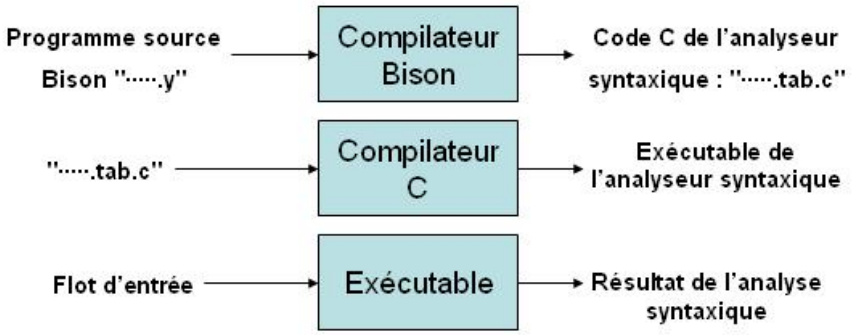

FIG. 7.1 – Création d'un analyseur syntaxique avec **Bison**.

Il est possible d'intégrer un analyseur lexical avec l'analyseur syntaxique généré par **Bison**, soit en utilisant le compilateur **Flex**, soit en le programmant directement en C. Dans le cas d'un analyseur lexical crée par **Flex**, il faut compiler le fichier de spécification avec l'option "-d" afin de donner à **Bison** la possibilité de gérer automatiquement les **codes des unités lexicales**. Il faut également inclure le fichier "$\cdots.tab.h$" dans le code de l'analysur **Flex**. Dans le cas où l'analyseur

lexical a été écrit directment en C, alors il faut redéfinir la fonction **yylex**.

Supposons que le code Bison est écrit dans le fichier "*CodeBison.y*" et que le code Flex est écrit dans le fichier "*CodeFlex.lex*". Voici comment compiler pour créer l'analyseur syntaxique :

```
bison -d CodeBison.y
flex CodeFlex.lex
gcc -o Final CodeBison.tab.c lex.yy.c -ly -lfl
```

7.2 Étapes pour utiliser Bison

Pour écrire un analyseur avec **Bison**, le processus comprend les étapes suivantes :

1. Écrire la spécification formelle de la grammaire dans un format reconnu par **Bison**. Pour chaque règle de production du langage, décrire l'action à exécuter lorsqu'une instance de cette règle est reconnue. L'action doit être une séquence d'instructions C.
2. Écrire un analyseur lexical qui va reconnaître les lexèmes et les délivrer à l'analyseur syntaxique.
3. Écrire un contrôleur (un programme) qui appelle l'analsyeur produit par **Bison**.
4. Écrire les routines d'erreurs.

7.3 Format de spécification Bison

Un fichier **Bison** comprend **quatre sections** :

```
%{
/* Prologue : Déclarations C */
%}
/* Déclarations Bison */
%%
/* Règles de production de la grammaire */
%%
/* Epilogue : Code C auxilliaire */
```

▷ La **section des déclarations** C inclue :

- La déclaration des **types**, les **variables** et les **fonctions globales** nécessaires pour écrire les actions.
- L'**inclusion de fichiers d'en-tête** par la commande du pré-processeur #**include**.
- La **définition des macros** par #**define**.
- La **déclaration**, si nécessaire, des fonctions **yylex** (pour l'analyseur lexical écrit à la main) et **yyerror** (pour reporter les erreurs syntaxique).

▷ La **section des déclarations Bison** inclu :

- La déclaration des **noms des symboles terminaux** et **non-terminaux**.
- La description de la **précédence des opérateurs**.
- La déclaration des **types de données des valeurs sémantiques** des différents symboles.

▷ La **section des règles de production de la grammaire** décrit comment construire chaque symbole non-terminal à partir de ses composants.

▷ La **section du code** C **auxilliaire** contient n'importe quel code jugé utile pour l'analyseur, notamment la **définition des fonctions** déclarées dans la première section.

Remarque 7.3.1 *Les commentaires "*$/*...*/$*" et "*$//$*" peuvent être insérés dans n'importe quelle section.*

7.4 La section du prologue

Les éléments déclarés dans cette section seront **recopiés** dans le fichier de l'analyseur syntaxique en haut et avant la définition de la focntion **yyparse**. Si aucune déclaration C n'est nécessaire, on peut omettre les symboles '%{' et '%}'. On peut avoir plusieurs sections de prologue intermixées avec les déclarations **Bison**. Cela permet d'avoir des déclarations C et des déclarations **Bison** qui se refèrent les unes aux autres. Par exemple, dans le code suivant :

```
%{
#include <stdio.h>
#include "ptypes.h"
%}

%union {
long int n;
tree t; /* tree est défini dans "ptypes.h". */
}

%{
static void print_token_value(FILE*, int, YYSTYPE);
#define YYPRINT(F, N, L) print_token_value(F, N, L)
%}
```

la déclaration $\%$**union** peut utiliser les types définis dans les fichiers d'en-tête indiqués dans la première section du prologue, et favorise de prototyper les fonctions qui prennent des arguments de type **YYSTYPE**.

7.5 La section des déclarations Bison

7.5.1 Symboles terminaux

Les symboles **terminaux** (*lexèmes*, *tokens*) sont représentés dans **Bison** par des **codes numériques**. Ces codes sont retournés par la fonction **yylex** de l'analyseur lexical. On n'a pas besoin de connaître la valeur d'un code, mais seulement le nom du symbole utilisé pour l'indiquer. Pour donner un **nom** à un **symbole terminal**, on utilise la **clause** $\%$**token** :

```
%token NomSymbolique
```

Il y a trois moyens pour écrire des symboles terminaux :

1. Les **tokens nommés** : ils sont écrits en utilisant des identificateurs C. Par convention, on les écrit en utilisant des lettres majuscules (il s'agit d'une convention et non pas d'une règle stricte). Ils doivent être déclarés dans la section des déclarations **Bison** en utilisant la clause $\%$**token**.
2. Les **tokens caractères** : ils sont écrits en utilisant la même syntaxe de C pour écrire des constantes caractères. Par exemple, '$+$' et '$\backslash n$'. Par convention, un token caractère est employé uniquement pour représenter un token

qui consiste en un caractère individuel. Ainsi, '+' est utilisé pour représenter le caractère '+' comme lexème.

3. Les **tokens chaînes de caractères** : ils sont écrits comme en C en utilisant les guillemets. Par exemple, "<=" est un token sous forme d'une chaîne de caractères. On peut associer un token de ce type avec un nom symbolique comme un alias en utilisant la clause %**token** :

   ```
   %token INFOUEGAL "<="
   ```

   À défaut, l'analyseur lexical fait correspondre un code numérique au token chaîne de caractères en consultant la table **yytname**.

Remarque 7.5.1 *Les tokens caractères et chaînes de caractères n'ont pas besoin d'être déclarés, sauf s'il y a besoin de spécifier les types de données de leurs valeurs sémantiques, leurs associativités, ou leurs précédences.*

Bison convertira chaque déclaration %**token** en une directive #**define** dans l'analyseur syntaxique afin que la fonction **yylex** puisse utiliser le code numérique de chaque token. Les noms symboliques doivent être présents dans l'analyseur lexical. Par exemple, si l'on a écrit dans **Flex** :

```
[a-zA-Z_][a-zA-Z0-9_]*  return IDENT;
[0-9]+                  return NOMBRE;
```

alors, il faut écrire les déclarations suivantes dans **Bison** :

```
%token IDENT NOMBRE
```

Il est possible de définir soi-même un code numérique (décimal ou hexadécimal) pour un token :

```
%token NUM   300
%token XNUM  0x12d
```

Cependant, il est généralement bon de laisser **Bison** choisir les codes numériques pour tous les tokens.

7.5.2 Précédence des opérateurs

Bison offre des directives pour fixer la **précédence** et l'**associativité** pour les opérateurs :

- ▷ La directive %**left** permet de déclarer un opérateur d'**associativité gauche**.
- ▷ La directive %**right** permet de déclarer un opérateur d'**associativité droite**.
- ▷ La directive %**nonassoc** permet de déclarer un opérateur **non associatif**.

La priorité des différents operateurs est controllée par l'ordre de leurs déclarations. La première déclaration %**left** ou %**right** dans le fichier donne la priorité faible aux premiers opérateurs déclarés, la dernière déclaration %**left** ou %**right** donne la priorité forte aux derniers opérateurs déclarés. Les opérateurs déclarés dans une même ligne auront la même priorité.

Exemple 7.5.1

```
%left  '<' '>' '='
%left  '+' '-'
%left  '*' '/'
%right '^'  /* opérateur de puissance */
```

Dans cette exemple, tous les opérateurs ont une associativité gauche, sauf le dernier qui a une associativité droite. L'opérateur '$\wedge$' a la plus forte priorité et les trois premiers opérateurs '$<$', '$>$' et '$=$' ont la plus faible priorité.

Certains opérateurs ont plusieurs priorités. Par exemple, le signe '$-$' comme opérateur **unaire** possède une prioprité très forte, et une priorité plus faible en tant qu'opérateur binaire. Les directives %**left**, %**right** et %**nonassoc** ne peuvent pas être utilisées plusieurs fois avec les mêmes opérateurs. Pour traiter ce problème, **Bison** offre une quatrième directive : le modificateur de priorité %`prec`. Comme exemple, on va résoudre ce problème pour l'opérateur '$-$'. Tout d'abord, on déclarera une précédence pour un symbole terminal artificil, disons `UNMOINS`. Ce token est artificile, car il ne sera jamais délivré par l'analyseur lexical. Il servira tout simplement à régler la priorité. Ensuite, on déclarera l'opérateur '$-$' binaire en écrivant :

```
%left '-'
```

Après, on déclarera l'opérateur '$-$' unaire comme étant de même priorité que l'opérateur `UNMOINS` en écrivant :

```
%left UNMOINS
```

Maintenant la précédence de `UNMOINS` peut être utilisée dans la règle de production appropriée :

```
...
%left '+' '-'
%left '*' '/'
%left UNMOINS
...
expr : expr '-' expr
     | '-' expr %prec UNMOINS
...
```

La dernière ligne signifie que l'opérateur '$-$' dans le contexte de la règle :

$$expr \rightarrow - \; expr$$

a une priorité identique à celle de l'opérateur artificiel `UNMOINS`, donc plus grande que celle des opérateurs '$+$', '$-$' binaire, '$*$' et '$/$'.

7.5.3 Les symboles non-terminaux

Par convention, le nom d'un symbole **non-terminal** est écrit en minuscules. Le nom d'un symbole non-terminal est un identificateur C. Il ne faut pas donner aux symboles (terminaux ou non-terminaux) des identificateurs qui sont des mots-clé de C.

Par défaut, le **premier symbole non terminal** dans la grammaire est l'**axiome**. Si l'on veut spécifier un autre symbole comme axiome, on doit écrire une directive $\%$**start** dans la section des déclarations **Bison**. Par exemple, le code suivant :

```
...
%start prog
...
```

indique que le symbole non-terminal "`prog`" est l'axiome de la grammaire.

7.5.4 Typage des symboles

Par défaut, les symboles (**terminaux** et **non-terminaux**) sont de type **int**. Il est possible de le changer par un autre via la macro :

```
#define YYSTYPE
```

dans le prologue. Par exemple, si l'on veut que tous les symboles soient de type `double`, on ajoute la ligne suivante :

```
%{
#define YYSTYPE double
%}
```

Si les symboles ont des types différents, ce qui est normalement le cas en pratique, il faut déclarer tous ces différents types dans une **union** en utilisant la directive %**union** (dans la section des déclarations **Bison**). Dans ce cas, il est nécessaire de préciser le type de chaque symbole lors de sa déclaration en utilisant la directive %**type**. La syntaxe de cette déclaration est :

```
%type <Type_Name> Name
```

Il est également possible de déclarer le type d'un symbole terminal au moment de sa déclaration avec la directive %**token** selon la syntaxe suivante :

```
%token <Type_Name> Name
```

Supposons par exemple que l'on a trois symboles terminaux : `IDENT` dont la valeur (de son **attribut**) est de type chaîne de caractères, `NUMBER` dont la valeur est de type entier et `REAL` dont la valeur est de type double. En plus, on a un symbole terminal 'exp' de type double. Tout d'abord, on doit déclarer une union des trois types indiqués comme suit :

```
%union {
   int    intval;
   double dblval;
   char*  strval;
}  /* sans ; */
```

En suite, on passe aux déclarations des types, en écrivant par exemple :

```
%token <intval> NUMBER
%type  <dblval> REAL
%type  <strval> IDENT
%type  <dblval> exp
```

7.6 La section des règles de la grammaire

7.6.1 Écriture d'une grammaire Bison

La forme générale d'une **règle de production** dans **Bison** est :

```
tete : corps ;
```

où :

- ▷ '$tete$' est le symbole **non-terminal** qui définit le **membre gauche** la règle de production.
- ▷ '$corps$' est la séquence des symboles **terminaux** et **non-terminaux** qui définissent le **membre droit** de cette règle de production.

Le point virgule '$;$' à la fin indique que la définition de la règle de production est terminée.

Exemple 7.6.1 *La règle de production :*

```
exp : exp '+' exp ;
```

signifie que deux expressions séparées par le symbole '$+$', peuvent être combinées pour former une autre expression.

Remarque 7.6.1 *Les caractères **espaces** dans les règles sont utilisés uniquement pour séparer les symboles.*

Les règles de production issues d'un même symbole non-terminal peuvent être définies séparamment (une par une) ou joingées en utilisant le caractère barre-verticale '$|$' comme suit :

```
tete : corps1
     | corps2
     | ...
     ;
```

Pour écrire une règle de production avec un membre droit qui est le symbole ε, il suffit de laisser vide le corps de cette règle.

Exemple 7.6.2 *Pour définir une séquence d'expressions formée par* 0, 1, *ou plusieurs expressions séparées par des virgules, on écrira :*

```
seqexp :
       | seqexp1
       ;
seqexp1 : exp
        | seqexp1 ',' exp
        ;
```

7.6.2 Actions

Une **action** accompagne une règle de production et contient du code C qui sera exécuté chaque fois qu'une instance de cette règle vient d'être reconnue. À travers une action associée à une règle, il est posssible de calculer une **valeur sémantique** d'un symbole de cette règle.

Une action consiste en une ou plusieurs instructions C entourées par '$\{$' et '$\}$' (qui sont obligatoires, même s'il s'agit d'une seule instruction). Elle peut être placée à n'importe quel endroit dans le corps d'une règle, et est exécutée à cet endroit. La plupart des actions ont juste une seule action à la fin de la règle.

Le code C dans une action peut se référer aux **valeurs sémantiques** des composants qui ont déjà été reconnus en utilisant la **pseudo-variable** '$\$i$', qui signifie la **valeur sémantique du i-ème composant** de la règle. La valeur sémantique du tête de la règle est référencée par la pseudo-variable '$\$\$$'. Lorsque **Bison** copie les actions dans le fichier de l'analyseur syntaxique, il traduit chacune de ces variables en une expression du type approprié. La **pseudo-variable** '$\$\$$' est traduit à une **lvalue modifiable**, donc elle peut employée à gauche d'une opération d'affectation. Ainsi, la règle :

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

reconnaît une expression (le non-terminal exp) comme étant une expression (le premier non-terminal exp juste après '$:$'), suivie du signe '$+$', suivi d'une expression (le second non-terminal exp). L'action asociée délivre (dans la pseudo-variable '$\$\$$') la valeur de la somme des valeurs des deux expressions référencées par '$\$1$' et '$\$3$'.

Remarque 7.6.2 *Attention, le token '$+$' est également compté comme un composant de la règle (il est référencé par la pseudo-variable '$\$2$').*

Si aucune action n'est spécifiée pour une règle, **Bison** lui associe l'action par défaut :

```
$$ = $1;
```

Bien évidemment, l'action par défaut n'est valide que lorsque la **tête** et le **premier composant** de la règle **ont le même type**. Il n'y a pas d'action par défaut pour une règle ayant un corps vide.

7.6.3 La variable globale yylval

La valeur sémantique d'un token doit être stockée dans la **variable globale** prédéfinie **yylval**. Cette variable est partagée par les deux analyseurs : l'analyseur lexical l'utilise pour stocker les valeurs sémantiques des tokens et l'analyseur syntaxique l'utilise pour consulter ces valeurs. Lorsque toutes les valeurs sémantiques sont tous du même type, la variable **yylval** sera également de ce type. Ainsi, si le type est "**int**" (type par défaut), on doit écrire le code suivant dans l'analyseur lexical (ou dans **yylex**) :

```
...
yylval = valuer; return NOMBRE;
/* On met valeur dans la pile Bison,
   puis on retourne le code du token. */
...
```

Lorsqu'on utilise plusieurs types pour les valeurs sémantiques, le type de **yylval** est l'union déclarée en utilisant la directive %**union**. Par conséquent, lorsqu'on veut stocker la valeur d'un token, on doit utiliser le nom adéquat du champ de l'union. Si par exemple, on a déclaré l'union comme suit :

```
%union {
   int    intval;
   double dblval;
}
```

alors, le code dans `yylex` doit être le suivant :

```
...
yylval.intval = valuer; return NOMBRE;
/* On met valeur dans la pile Bison,
   puis on retourne le code du token. */
...
```

7.7 La section de l'épilogue

Le contenu de cette section est recopié texto en fin du fichier de l'analyseur. Si cette section est vide, on peut omettre le second symbole '%%'. Comme l'analyseur **Bison** contient lui-même plusieurs macros et plusieurs identificateurs qui commencent par 'yy' or 'YY', alors une bonne idée est de ne pas utiliser de tels noms dans l'épilogue, à l'exception bien sûr ceux cités dans le manuel d'utilisation de **Bison**.

7.8 La fonction yyparse

L'analyseur syntaxique construit par **Bison** contient une fonction C nommée **yyparse**$()$. Lorsque cette fonction est appelée, elle met l'**analyseur syntaxique** en marche. Elle lit les tokens délivrés par l'analsyeur lexical, exécute les actions des règles reconnues, et se termine lorsqu'elle lit le token représentant la **fin de de l'entrée** ou une **erreur de syntaxe non recouvrable**. Le prototype de cette fonction est :

```
int yyparse (void);
```

Selon l'état de l'analyse syntaxique, la fonction `yyparse` retourne :

- ▷ 0, si l'analyse syntaxique se termine avec **succès** $($le token fin de l'entrée a été rnecontré$)$.
- ▷ 1, si l'analyse syntaxique se termine avec un **échec** $($une erreur de syntaxe, ou une macro **YYABORT** a été exécutée$)$.
- ▷ 2, si l'analyse syntaxique se termine par une erreur causée par la **mémoire**.

Il est possible dans une action associée à une règle, de forcer la fonction **yyparse** à retourner une valeur en utilisant ces macros :

- ▷ **YYACCEPT** : retourner immédiatemment la valeur 0 $($reporter un succès$)$.
- ▷ **YYABORT** : retourner immédiatemment la valeur 1 $($reporter un échec$)$.

7.9 Traitement des erreurs de syntaxe

Lorsque l'analyseur produit par **Bison** rencontre une erreur, il appelle par défaut la fonction **yyerror**$($**char**$*)$ qui se contente d'afficher le message "**syntax error**" puis il s'arrêete. Cette fonction peut être redéfinie par le programmeur.

Dans **Bison**, il est possible de définir la façon de **recouvrir une erreur** de syntaxe $($**récupération sur erreur**$)$ en écrivant des règles qui reconnaissent un **token spécial** : "**error**". C'est un symbole terminal **prédéfini** et réservé pour le traitement des erreurs.

En effet, l'analyseur syntaxique produit par **Bison** génère automatiquement un token **error** chaque fois qu'une erreur de syntaxe survienne. Si l'on a fournit une

règle pour reconnaître ce token dans le contexte courant, l'analyseur va continuer son travail. On peut rajouter dans toute production de la forme $A \rightarrow \alpha_1 \mid ... \mid \alpha_n$, une production de la forme :

$$A \rightarrow \textbf{error}\ \beta$$

Dans ce cas, une règle de récupération sur erreur sera traitée comme une règle de production classique. On pourra donc lui associer une action sémantique contenant un message d'erreur. Dès qu'une erreur est rencontrée, tous les symboles sont avalés jusqu'à rencontrer le token correspondant à β. Par exemple :

```
instr : error ';'
```

indique à l'analyseur qu'à la vue d'une erreur, il doit sauter jusqu'au delà du prochain symbole ';' et supposer qu'un '$instr$' vient d'être reconnue. La routine **yyerrok** replace l'analyseur dans le mode normal de fonctionnement c'est-à-dire que l'analyse syntaxique n'est pas interrompue.

7.9.1 Un exemple

On va illustrer ce traitement des erreus en montrant un analyseur destiné à vérifier interactivement la saisie de données. Les données sont composées d'un nom suivi d'un entier sur une ligne. En sortie, les données de chaque ligne sont rangées de la façon suivante : l'entier, un symbole ':', et le nom. Voici le code **Bison** de cet analyseur syntaxique (Fichier "$parser.y$") :

```
%{
#include <stdio.h>
extern FILE* fp;
void yyerror(char*);
%}

%union {
   int intval;
   char* strval;
}

%token <intval> ENTIER
%token <strval> NOM
```

```
%%

axiome : entree { puts("C'est fini"); }
       ;

entree : ligne
       | entree ligne
       ;

ligne  : NOM ENTIER '\n' { fprintf(fp, "%d:%s\n", $2, $1); }
       | error '\n' { printf("Refaire : "); } ligne
       ;

%%

int main()
{
   fp = fopen("resultats.txt", "w");
   yyparse();
   fclose(fp);
}

void yyerror(char* mess)
{
   puts(mess);
}
```

Voici le code **Flex** de l'analyseur lexical (Fichier "$lexer.lex$") :

```
%{
#include "parser.tab.h"
#include <stdlib.h>
%}
%%
[a-z]+ { yylval.strval = strdup(yytext); return NOM; }
[0-9]+ { yylval.intval = atoi(yytext); return ENTIER; }
\n     return *yytext;
[ \t]  ;
.      { printf("%c: illegal\n", *yytext); }
```