Automates et langages

Support de cours

Jacques Malenfant
professeur des universités
© Jacques Malenfant, 2003

Licence SIR
UFR Sciences et Sciences de l'ingénieur
Université de Bretagne sud

Table des matières

1 Langages formels **5**

1.1 Les bases 5

1.2 Problèmes fondamentaux 6

1.3 Hiérarchie de Chomsky 7

2 Langages réguliers et automates finis **9**

2.1 Construction d'un langage régulier 9

2.2 Expressions régulières 10

2.3 Automates finis 10

2.4 Mise en pratique des algorithmes 17

2.4.1 Construction de Thompson 17

2.4.2 Construction d'un AFD à partir d'un AFN 19

2.4.3 Construction directe de l'AFD à partir de l'expression régulière 22

2.4.4 Minimisation d'un AFD 25

2.5 Exercices 26

3 Application des expressions régulières **27**

3.1 L'outil `egrep` 27

3.2 Autres commandes et outils Unix 30

3.2.1 La commande `find` 30

3.2.2 La commande `sed` 31

3.2.3 L'éditeur `emacs` 31

3.3 Les expressions régulières de Perl 32

3.4 Le modèle objet de Python et Java 34

4 Langages non-contextuels, grammaires et automates à pile **39**

4.1 Grammaires non-contextuelles 39

4.2 Automate à pile 42

4.3 Transformations de grammaires 44

4.3.1 Factorisation à gauche 44
4.3.2 Elimination des récursivités à gauche 45

5 Le langage XML 49
5.1 Langage à balises sémantiques 49
5.2 Types de documents et DTD 52
5.3 DTD et grammaires 55
5.4 Les schémas XML 58

6 Construction des analyseurs pour grammaires non-contextuelles 59
6.1 Analyse des langages non-contextuels 59
6.2 Analyse descendante 61
6.2.1 Idée générale de l'analyse descendante 61
6.2.2 Descente récursive 63
6.2.3 Analyseurs prédictifs non-récursifs de type LL(1) 71
6.3 Analyse ascendante 74
6.3.1 Algorithme d'analyse général 75
6.3.2 Construction des tables d'analyse ascendante de type LR 78

7 Analyse sémantique, grammaires attribuées et transformations 83
7.1 Définitions dirigées par la syntaxe 83
7.2 Schéma de traduction 87
7.2.1 Élimination des actions intérieures en faveur d'actions à la réduction . . 89
7.2.2 Héritage des attributs dans la pile d'analyse 89

8 Transformations de documents XML 93
8.1 Le langage de transformation XSL 93
8.2 Grammaires attribuées et transformations 101
8.3 API DOM et la transformation de documents 107
8.3.1 Définition de l'API DOM 107
8.3.2 Parcours de l'arbre du document 109

Bibliographie 119

Chapitre 1

Langages formels

Dans ce chapitre, nous introduisons les notions fondamentales des langages formels de même que le problème de définition et de reconnaissance des langages.

1.1 Les bases

Comme dans la langages naturels (français, anglais, russe, ...), la base de tout langage est formée des signes élémentaires à partir desquels seront formés les mots et les phrases de ce langage. Dans le domaine des langages formels, ces signes élémentaires s'appellent *symboles* :

symbole : signe élémentaire (lettre, chiffre, ...).

L'ensemble des symboles qui peuvent être utilisés dans un langage donné s'appelle un *alphabet* :

alphabet : (souvent noté Σ) ensemble fini de symboles.

Là où dans les langages naturels courants nous parlons de mots et de phrases, les langages formels ne parlent que de *chaînes* :

chaîne : séquence finie de symboles choisis parmi un alphabet.

En tant que séquences de symboles, l'ordre dans lequel apparaissent les symboles dans les chaînes est significatif. Il devient donc possible de parler de notions de préfixe ou de suffixe pour désigner les premiers et les derniers symboles d'une chaîne :

préfixe d'une chaîne : séquence de symboles apparaissant au début de la chaîne.

suffixe d'une chaîne : séquence de symboles apparaissant à la fin de la chaîne.

L'ordre des symboles influe également sur l'opération de construction de chaînes à partir d'autres chaînes par juxtaposition que l'on appelle *concaténation* :

concaténation : notée '$\cdot$', c'est l'opération par laquelle deux chaînes sont combinées par juxtaposition faisant d'abord apparaître les symboles de la première chaîne (dans le même ordre) puis les symboles de la seconde chaîne (toujours dans le même ordre).

Exemple 1.1 Soir $\omega_1 = ab$ et $\omega_2 = cd$, alors la concaténation de ω_1 avec ω_2, notée $\omega_1 \cdot \omega_2$, est égale à $abcd$. □

Par ailleurs, il est souvent utile de pouvoir déterminer la longueur d'une chaîne :

> **longueur d'une chaîne ω :** (notée $|\omega|$) nombre de symboles apparaissant dans la chaîne.

Compte tenu de la propriété de longueur, il existe une chaîne qui se distingue de toute les autres par le fait qu'elle a longueur 0, la *chaîne vide* :

> **chaîne vide :** notée ϵ, c'est la chaîne contenant aucun symbole, de longueur 0 (c'est-à-dire $|\epsilon| = 0$).

La notion de langage formel est en réalité extrêmement simple : un *langage formel* est simplement un ensemble de chaînes définies sur un certain alphabet :

> **langage formel :** ensemble de chaînes définies sur un certain alphabet Σ.

Exemple 1.2 Soit un alphabet $\Sigma = \{0, 1\}$, on peut définir le langage L des séquences de 0 et 1 formant des palindrômes, c'est-à-dire que l'on peut les lire indifféremment de gauche à droite ou de droite à gauche. □

La relation entre alphabet et langage est consubstantielle : sans alphabet, pas de langage. À partir d'un alphabet donné, tout ensemble de chaînes formé sur cet alphabet est un langage. La notion d'inclusion ensembliste nous permet de comparer les langages : on peut parler d'un langage L_1 qui serait inclus dans un langage L_2, c'est-à-dire que toutes les chaînes du langage L_1 apparaissent dans le langage L_2. Parmi tous les langages que l'on peut former en combinant 0, 1 ou plusieurs symboles d'un certain alphabet, il en est un particulier dans lequel tous les autres langages sont inclus : le *langage total* :

> **Σ^* :** ensemble de toutes les chaînes que l'on peut former sur l'alphabet Σ, y compris la chaîne vide ϵ.

On peut noter que le langage Σ^* est formé de la combinatoire en position et en longueur de tous les symboles de l'alphabet Σ, et donc que tout langage formé sur Σ est nécessairement un sous-ensemble de Σ^*. C'est pour cette raison que l'on appelle Σ^* le *langage total* sur Σ. Le langage complet contient bien sûr un nombre infini de chaînes, puisque les chaînes ne sont pas bornées en longueur. Pourtant, il est assez facile d'énumérer ce langage : on commence par la chaîne vide, puis les chaînes de longueur conteant chacun des symboles de l'alphabet, puis on passe aux chaînes de longueur 2, etc.

En plus du langage complet, on peut définir d'autres langages particuliers comme le langage vide sur Σ, $L = \emptyset$ qui ne contient aucune chaîne. Il ne faut pas confondre le langage vide avec le langage qui ne contient que la chaîne vide, $\{\epsilon\}$, qui lui n'est pas vide mais contient une seule chaîne, la chaîne vide.

1.2 Problèmes fondamentaux

Deux problèmes fondamentaux se posent dans le domaine des langages formels, et ces deux problèmes sont en réalité intimement liés :

1. Comment définir un langage en intension ?
2. Étant donné un langage L sur Σ, comment décider (mécaniquement) si une chaîne $\omega \in \Sigma^*$ appartient à L ou non ?

En effet, les langages sont des ensembles. La manière la plus simple de définir un ensemble est en extension : il suffit de lister toutes les chaînes appartenant à ce langage. Cette approche n'est bien entendu possible que si le langage est *fini*, c'est-à-dire s'il contient un nombre fini de chaînes. Les chaînes constructibles n'étant pas bornées en longueur, on peut construire une infinité de chaîne sur un alphabet ; les langages intéressants contiennent donc le plus souvent un nombre infini de chaînes.

Si un langage contient un nombre infini de chaînes, il devient impossible de le définir en extension. Il faut donc passer à une définition en intension. L'exemple des palindrômes sur $\Sigma = \{0, 1\}$ fait appel à une définition en intension, bien que celle que nous avons donnée ne soit pas très formelle. En fait, la propriété de se lire indifféremment de gauche à droite ou de droite à gauche donne un critère permettant de décider si une chaîne de Σ^* appartient ou non au langage des palindrômes sur Σ.

Nous voici donc déjà sur les plates-bandes du second problème : comment décider si une chaîne appartient au langage. Avec la définition précédente, vous pouvez tous décider si une chaîne appartient au langage des palindrômes ou non :
- 01010 est un palindrôme, alors que
- 01100 n'en est pas un.

Certes la faculté de l'humain d'interpréter les définitions en langage naturel pour décider est très puissante. Ce qui intéresse l'informaticien, c'est plutôt de savoir s'il peut écrire un programme qui puisse lire une chaîne et décider si cette chaîne appartient ou non à un certain langage. Ce problème est analogue à celui de décider si une phrase en français est grammaticalement et orthographiquement correcte. Il est aussi analogue, comme nous le verrons plus loin, à celui de décider si un programme écrit dans un certain langage de programmation est syntaxiquement correct ou non.

Une façon de résoudre le second problème consiste donc à construire ce que l'on va appeler un *accepteur* pour un langage L : une procédure mécanique qui, lorsqu'on lui soumet une chaîne ω, répond vrai si $\omega \in L$ et faux sinon.

1.3 Hiérarchie de Chomsky

Comment construire cet accepteur ? Aussi surprenant que cela puisse paraître, la capacité à construire un tel accepteur dépend de la complexité de la structure des chaînes de L et il existe en fait des langages pour lesquels nous ne savons pas construire d'accepteurs. Mieux, il est intéressant de caractériser les langages en fonction de la «difficulté» de sa reconnaissance, difficulté qui se reflète dans la quantité de ressources (mémoire, temps, ...) nécessaires pour cette reconnaissance.

Certains langages sont très faciles à reconnaître. Le langage vide, ou le langage qui ne contient que la chaîne vide sont très facile à reconnaître. Le langage complet Σ^*, bien qu'infini,

est aussi très facile à reconnaître, puisqu'à partir du moment où une chaîne ne contient que des symboles de l'alphabet Σ, elle appartient à Σ^*. Bien sûr, tous les langages finis c'est-à-dire ne contenant qu'un nombre fini de chaînes, sont faciles à reconnaître puisqu'il suffit de mémoriser (dans un quantité finie de mémoire) toutes les chaînes acceptables.

Ce sont donc les langages infinis qui posent problème. En réalité, la facilité ou la difficulté de la reconnaissance dépend de la structure des chaînes admissibles. Les accepteurs de certains langages ne nécessitent par exemple qu'une quantité finie de mémoire, même si ces langages sont infinis. D'autres nécessitent une mémoire non-bornée gérée comme une pile. D'autres encore nécessitent une mémoire non-bornée à accès aléatoire («RAM»).

Noam Chomsky a proposé une hiérarchie des langages en fonction de la plus petite quantité de ressources nécessaires à leurs accepteurs. Quatre grandes catégories sont définies :

1. les langages rationnels ou réguliers,
2. les langages indépendants du contexte,
3. les langages dépendants du contexte, et
4. les langages récursivement énumérables.

Le théorème de Chomsky, établissant la hiérarchie, dit que les langages réguliers sont inclus dans les langages indépendants du contexte, qui sont eux-mêmes inclus dans les langages dépendants du contexte, ces derniers étant finalement inclus dans les langages récursivement énumérables.

Les langages rationnels ou réguliers ont une structure relativement simple et nous allons les étudier au prochain chapitre. Leurs accepteurs, appelés automates d'états fini, ne nécessitent qu'une quantité finie de mémoire et leur reconnaissance peut se faire de manière très efficace. Les langages indépendants du contexte qui ne sont pas réguliers nécessitent des reconnaisseurs disposant d'une pile pour mémoriser le contexte de retour après la reconnaissance du patron courant ; leur reconnaissance peut se faire de manière relativement efficace, surtout pour certaines classes parmi ces langages que nous étudierons plus loin également.

Les langages dépendants du contexte qui ne sont pas indépendants du contexte nécessitent plus de ressources encore dans la mesure où la reconnaissance d'une partie de chaîne nécessite non seulement de conserver le contexte vers lequel on doit revenir après la reconnaissance, mais nécessite aussi l'accès à des informations sur le contexte dans lequel se situe cette partie de chaîne. Enfin, les langages récursivement énumérables sont des langages pour lesquels on peut définir une procédure permettant d'énumérer récursivement toutes les chaînes appartenant au langage. Les langages récursivement énumérables qui ne sont pas dépendants du contexte sont dits *semi-décidables* puisque la procédure d'énumération nous offre un moyen simple de vérifier si une chaîne appartient au langage : énumérer toutes les chaînes jusqu'à rencontrer la chaîne à vérifier. Malheureusement, cette procédure d'acceptation ne permet pas de décider si une chaîne n'appartient pas au langage : le langage étant infini, il ne sera jamais possible de savoir si la chaîne qui n'a pas encore été obtenue le sera plus tard dans l'énumération ou jamais. L'accepteur peut donc répondre oui si la chaîne appartient au langage, mais peut boucler indéfiniment si la chaîne n'appartient pas au langage.

Chapitre 2

Langages réguliers et automates finis

Les langages formels dont la structure est la plus simple sont les langages dits *réguliers* ou *rationnels*. Il suffit pour reconnaître les chaînes appartenant à un de ces langages de construire un automate fini. Nous allons voir dans ce chapitre comment sont définis ces langages, comment on peut construire un automate fini reconnaissant les chaînes appartenant à un langage à partir de sa définition.

2.1 Construction d'un langage régulier

Rappelons que les langages formels sont des ensembles de chaînes construites sur un alphabet donné. Rappelons également la question importante de savoir comment définir en intension un langage formel. Les langages réguliers sont des langages qui peuvent se définir à partir d'un nombre fini de langages de base eux-mêmes finis, et d'opérations ensemblistes et de concaténation entre ces langages et entre les chaînes appartenant à ces langages.

Considérons un alphabet $\Sigma = \{a_1, ...a_n\}$. Définissons les langages *triviaux* (car finis et même singleton) $L_i = \{a_i\}$. Définissons ensuite les trois opérations suivantes :

union : $L \cup L' = \{\omega \mid \omega \in L \; ou \; \omega \in L'\}$.

produit : $L \cdot L' = \{u \cdot v \mid u \in L \; et \; v \in L'\}$.

itération (fermeture de Kleene) : $L^* = \bigcup_{i \geq 0} L_i$, où

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= L \cdot L^1 \\ &\dots \\ L^{i+1} &= L \cdot L^i \end{aligned}$$

et la fermeture positive $L^+ = \bigcup_{i > 0} L_i$

Ces seules opérations sont nécessaires pour construire les langages réguliers :

Langage régulier : Un langage $L \subseteq \Sigma^*$ est dit *régulier* (*rationnel*) si et seulement si il peut être obtenu à partir des langages triviaux sur les symboles de Σ uniquement par l'application des opérations d'union, de produit et de fermeture.

2.2 Expressions régulières

Pour construire un langage régulier sur un alphabet Σ, il suffit donc de pouvoir construire les langages triviaux à partir des symboles de Σ puis de combiner les langages par des opérations d'union, de produit et de fermeture. Une expression régulière est une expression à la manière des expressions arithmétique mais dont la signification se comprend en termes d'opérations sur les langages. Les expressions régulières sont donc définies à l'aide d'opérateurs représentant les opérations précédentes sur les langages et de manière inductive pour refléter leur structure récursive.

Expression régulière : (par induction) Soit Σ un alphabet, alors
- $\emptyset$ est une expression régulière dont la signification est $\mathcal{L}(\emptyset) = \emptyset$
- $\mathtt{a} \in \Sigma$ est une expression régulière dont la signification est $\mathcal{L}(\mathtt{a}) = \{\mathtt{a}\}$
- Soit e une expression régulière dénotant $\mathcal{L}(e)$, alors $e\mathtt{*}$ est aussi une expression régulière dénotant le langage $\mathcal{L}(e)^*$
- Soient e_1 et e_2 des expressions régulières dénotant les langages $\mathcal{L}(e_1)$ et $\mathcal{L}(e_2)$, alors
  e_1e_2 est une expression régulière dénotant le langage $\mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$
  $e_1\mathtt{+}e_2$ est une expression régulière dénotant le langage $\mathcal{L}(e_1) \cup \mathcal{L}(e_2)$

Les expressions régulières nous donnent un premier moyen pour exprimer des langages formels en intension.

2.3 Automates finis

Un automate fini est un modèle formel de calcul dont le comportement est très simple. On le représente le plus souvent graphiquement avec des nœuds qui représentent les états de l'automate et des flèches étiquetés par un symbole qui représentent des transitions possibles à partir d'un certain état de l'automate vers un autre lorsque le prochain symbole dans la chaîne à vérifier correspond à l'étiquette sur la flèche. Parmi les états de l'automate fini, on distingue un état initial et un ensemble d'états finaux. Si en partant de l'état initial, on arrive à un état final par un suite de transitions consommant un à un tous les symboles de la chaîne d'entrée, alors la chaîne est dite acceptée par l'automate.

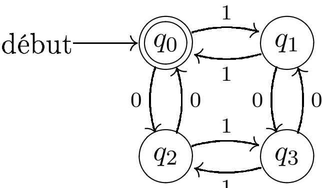
Automate fini déterministe (AFD) : un *automate fini déterministe* est une structure $(Q, \Sigma, \delta, q_0, F)$ telle que :
- Q est un ensemble fini d'états,
- Σ est un alphabet d'entrée (fini),
- $\delta : Q \times \Sigma \to Q$ est une fonction partielle de transition qui, étant donné un état $q_i \in Q$ et un symbole $a \in \Sigma$, retourne un nouvel état $q_j \in Q$,
- $q_0 \in Q$ est l'état initial de l'automate, et
- $F \subseteq Q$ est l'ensemble des états finaux.

Comportement accepteur : Soit $(Q, \Sigma, \delta, q_0, F)$ un automate fini. Si à partir de q_0, en appliquant successivement les transitions définis par δ sur chacun des symboles d'une chaîne ω dans l'ordre, on arrive dans un état $f \in F$, alors la chaîne ω est dite *acceptée* par l'automate.

Exemple 2.1 Considérons l'automate $A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$ dont la fonction de transition δ est définie par la table suivante :

États	Entrées 0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Cet automate est représenté ci-dessous dans sa forme graphique, c'est-à-dire sous la forme d'un graphe orienté étiqueté. L'état initial q_0 est signalé par l'étiquette début et la flèche correspondante. Les transitions sont représentées par une flèche dont l'origine donne l'état de départ, l'étiquette l'entrée consommée, et la destination l'état vers lequel nous amène la fonction de transition. Les états finaux sont représentées par un double cercle.



Cet automate accepte la chaîne $\omega = 110101$, puisqu'en partant de l'état q_0, la consommation du premier symbole 1 nous amène dans l'état q_1. Le second symbole 1 nous ramène immédaitement de l'état q_1 vers l'état q_0. La consommation des deux symboles suivants 0 puis 1 nous amène de l'état q_0 vers l'état q_2 puis q_3. En consommant ensuite les deux derniers symboles 0 puis 1, on revient d'abord dans l'état q_1, puis dans q_0. La chaîne est alors entièrement consommée, et l'automate est dans l'état q_0 qui est un état final. On accepte donc la chaîne 110101. □

Le comportement accepteur de l'automate nous amène à parler de la notion de langage accepté par un automate :

> **Langage accepté :** le langage accepté par un automate fini A, noté $\mathcal{L}(A)$, est l'ensemble des chaînes ω acceptées par A, c'est-à-dire $\mathcal{L}(A) = \{\omega \mid \omega \text{ est acceptée par } A\}$.

Ainsi, le langage accepté par un automate fini est un langage formel, puisqu'il s'agit d'un ensemble de chaînes sur l'alphabet d'entrée de l'automate. En fait, l'automate donne une définition dite *opérationnelle* de ce langage, dans la mesure où il définit une procédure de décision pour distinguer les chaînes du langage $\mathcal{L}(A)$ des autres chaînes constructibles sur Σ^*.

Les automates qui sont conformes à la définition précédente sont dits *déterministes* car la fonction de transition est telle que pour chaque état, sur une entrée donnée, on se retrouve dans au plus un état (la fonction de transition est partielle, il est donc possible qu'il n'y ait pas de transition définie pour un état sur une entrée donnée). Les théoriciens ont également défini une forme que d'automate que l'on croyait a priori plus puissante, l'automate fini non-déterminste. Un automate fini non-déterministe permet plusieurs transitions vers différents états depuis un état donné et sur un symbole donné. De plus, il permet des transitions spontanées, sans consommation de symbole de l'entrée, appelée ϵ-transition.

> **Automate fini non-déterministe (AFN) :** un *automate fini non-déterministe* est une structure $(Q, \Sigma, \delta, q_0, F)$ telle que :
> – Q est un ensemble fini d'états,

Construction d'une expression régulière à partir d'un AFD.

Soit un AFD $M = (Q, \Sigma, \delta, q_0, F)$. L'idée est de construire par induction les expressions régulières r_{ij}^k dénotant la séquence de symboles de Σ permettant de faire passer l'AFD M de l'état i à l'état j sans passer par un état d'indice plus grand que k.
Soit R_{ij}^k le langage reconnu par l'expression régulière r_{ij}^k, la définition récursive de R_{ij}^k est :

$$R_{ij}^0 = \begin{cases} \{a|\delta(q_i, a) = q_j\} & \text{si } i \neq j, \\ \{a|\delta(q_i, a) = q_j\} \cup \{\epsilon\} & \text{si } i = j, \end{cases}$$

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$$

Les expressions régulières r_{ij}^k dénotant ces langages sont obtenus simplement en appliquant les opérateurs correspondant aux opérations faites sur les langages, c'est-à-dire :

$$r_{ij}^0 = a_1 + ... + a_p \qquad si\ i \neq j$$

$$r_{ij}^0 = a_1 + ... + a_p + \epsilon \qquad si\ i = j$$

$$r_{ij}^k = r_{ik}^{k-1}(r_{kk}^{k-1})^* r_{kj}^{k-1} + r_{ij}^{k-1}$$

où $\{a|\delta(q_i, a) = q_j\} = \{a_1, ..., a_p\}$.

FIG. 2.1 – Construction d'une expression régulière à partir d'un AFD.

- Σ est un alphabet d'entrée (fini),
- $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$ est une fonction partielle de transition qui, étant donné un état $q_i \in Q$ et un symbole $a \in \Sigma$ ou ϵ, retourne un sous-ensemble d'états possibles $Q' \subseteq Q$,
- $q_0 \in Q$ est l'état initial de l'automate, et
- $F \subseteq Q$ est l'ensemble des états finaux.

L'utilisation d'un automate fini non-déterministe est en fait bien théorique. En effet, sur une chaîne donnée, l'exécution des transitions nécessitent à chaque étape ce que l'on appelle un choix non-déterministe entre les états suivants possibles (y compris ceux accessibles par ϵ-transition). On suppose qu'un tel choix nous amène à chaque fois vers un état tel que si la chaîne doit être acceptée, on arivera bel et bien dans un état final. Les américains, bien pragmatiques, appeleraient cela du «*wishful thinking*», ce qui peut s'apparenter à notre «prendre ses désirs pour des réalités». Comment en effet réaliser concrètement un mécanisme qui devant plusieurs alternatives choisiraient toujours la bonne ? Il n'existe pas de fait de manière de réaliser cela mécaniquement (informatiquement) aujourd'hui. L'automate fini non-déterministe est donc un modèle de calcul purement théorique.

Les questions intéressantes qui se posent maintenant consistent à savoir d'une part quelle est la relation entre les langages formels définis par les expressions régulières, ceux qui sont définis par les automates d'état finis déterministes et ceux enfin définis par les automates finis non-déterministes. Y a-t-il parmi ces formalismes un qui soit plus puissant que les autres ? La réponse est non. ces trois formalismes permettent de définir exactement la même classe de langages formels, les langages réguliers.

Construction d'un AFN à partir d'une expression régulière.

1. Pour l'expression régulière ϵ, construire l'automate $N(\epsilon)$ avec deux états : un état initial i et un état final f, et y définir une seule transition sur ϵ de i vers f. $N(\epsilon)$ reconnaît le langage $L = \{\epsilon\}$.
2. Pour l'expression régulière a, où $a \in \Sigma$, construire l'automate $N(a)$ avec deux états : un état initial i et un état final f, et y définir une seule transition sur a de i vers $f.N(a)$ reconnaît le langage $L = \{a\}$.
3. Soient $N(s)$ et $N(t)$ deux AFN pour les expression régulières s et t, alors :
   (a) Pour l'expression régulière $s|t$, construire l'automate $N(s|t)$ avec un état initial i avec une transition sur ϵ vers les états initiaux de $N(s)$ et $N(t)$, et un état final f avec une transition sur ϵ de chacun des états finaux des deux automates $N(s)$ et $N(t)$. L'automate $N(s|t)$ reconnaît ainsi le langage $L(s|t) = L(s) \cup L(t)$.
   (b) Pour l'expression régulière st, construire l'automate $N(st)$ avec comme état initial i l'état inital de $N(s)$ et comme état final l'état final de $N(t)$; enfin, on fusionne l'état final de $N(s)$ avec l'état initial de $N(t)$, et cet état fusionné perd son statut d'état final et initial dans $N(st)$. L'automate $N(st)$ reconnaît ainsi le langage $L(st) = L(s)L(t)$.
   (c) Pour l'expression régulière s^*, construire l'automate $N(s^*)$ avec un état initial i et un état final f ; ajouter une transition sur ϵ de i vers l'état initial de $N(s)$ et vers f ; ajouter aussi une transition sur ϵ de l'état final de $N(s)$ vers f ; enfin, ajouter une transition sur ϵ de l'état final de $N(s)$ vers l'état initial de $N(s)$. L'automate $N(s^*)$ reconnaît ainsi le langage $L(s^*) = \bigcup_{i=0}^{\infty} L(s)^i$.

FIG. 2.2 – Construction de Thompson.

Théorème 2.1 *(équivalence AFD/ER) Soit L un langage accepté par un AFD A, alors il existe une expression régulière e telle que $L(A) = L(e)$.*

Preuve. La preuve est par construction. On montre d'une part comment construire un automate fini déterministe à partir d'une expression régulière (voir la figure 2.3). On montre ensuite comment construire une expression régulière à partir d'un automate fini (voir la figure 2.1). *CQFD.*

Théorème 2.2 *(équivalence AFD/AFN) Soit L un langage accepté par un AFN A, alors il existe un AFD A' tel que $L(A) = L(A')$.*

Preuve. Encore une fois la preuve est par construction. Par définition, tout automate fini déterministe peut être tansformé en automate fini non-déterministe où la fonction de transition retourne toujours un singleton ou un ensemble d'états vide. On montre ensuite comment transformer tout automate fini non-déterministe en automate fini déterministe. L'idée de la construction est la suivante. L'automate déterministe est obtenu en prenant comme états des sous-ensembles d'états de l'automate non-déterministe. Les transitions de l'AFD simulent en parallèle toutes les transitions qui étaient possibles entre les états de l'AFN inclus dans un certain état de l'AFD en produisant comme nouvel état le sous-ensemble de tous les états atteints dans l'AFN. *CQFD.*

Construction directe de l'AFD (sans passer par un AFN).

Soit r une expression régulière, l'algorithme général est :
1. Construire un arbre abstrait pour l'expression régulière (r)#
2. Étiqueter les nœuds de cet arbre abstrait contenant des symboles d'entrée
3. Calculer les quatre fonctions *Annulable*, *PremièrePos*, *DernièrePos* et *PosSuivante*
4. Construire l'AFD à l'aide de la fonction *PosSuivante*

Calcul des fonctions *Annulable*, *PremièrePos* et *DernièrePos*
Les fonctions *Annulable* et *PremièrePos* sont définies par induction sur les nœuds de l'arbre abstrait de l'expression régulière selon les règles suivantes :

Nœud n	*Annulable(n)*	*PremièrePos(n)*
feuille étiquetée ϵ	**vrai**	$\emptyset$
feuille étiquetée i	**faux**	$\{i\}$
nœud étiqueté | de fils c_1 et c_2	*Annulable(c_1)* **ou** *Annulable(c_2)*	*PremièrePos(c_1)* $\cup$ *PremièrePos(c_2)*
nœud étiqueté $\bullet$ de fils c_1 et c_2	*Annulable(c_1)* **et** *Annulable(c_2)*	**si** *Annulable(c_1)* **alors** *PremièrePos(c_1)* $\cup$ *PremièrePos(c_2)* **sinon** *PremièrePos(c_1)*
nœud étiqueté $*$ de fils c_1	**vrai**	*PremièrePos(c_1)*

DernièrePos est calculée comme *PremièrePos* en intervertissant les rôles de c_1 et c_2.

Calcul de la fonction *PosSuivante*
Deux règles suffisent à calculer la fonction *PosSuivante* :

1. Si n est un nœud étiqueté par $\bullet$ de fils c_1 et c_2 et si i est une position dans *DernièrePos(c_1)*, alors toutes les positions de *PremièrePos(c_2)* appartiennent à *PosSuivante(i)*.
2. Si n est un nœud étiqueté par $*$ et i est une position dans *DernièrePos(n)*, alors toutes les positions dans *PremièrePos(n)* sont dans *PosSuivante(i)*.

Construction de l'AFD.

1. *DÉtats* := {*PremièrePos(racine)*}
2. **tant que** il existe dans *DÉtats un état non-marqué T* ***faire***
3.   marquer T
4.   **pour** chaque symbole de l'alphabet $a \in \Sigma$ **faire**
5.   soit U l'ensemble des positions qui appartiennent à *PosSuivante(p)* pour une position p de T telle que le symbole en position p soit a.
6.   **si** U n'est pas vide et n'appartient pas à *DÉtats* **alors**
7.   ajouter U comme état non-marqué à *DÉtats*
8.   *DTran[T,a]* := U

    fin

fin

FIG. 2.3 – Construction directe d'un AFD à partir d'une expression régulière.

Construction d'un AFD à partir d'un AFN.

Calcul de $\epsilon\!-\!fermeture(T)$
Soit T un ensemble d'états de l'AFN, calculer l'ensemble des états comprenant T et tous les états atteignables depuis un état de T par une ou plusieurs transitions sur ϵ.

1. Empiler tous les états de T dans *pile*
2. initialiser $\epsilon\!-\!fermeture(T)$ à T
3. **tant que** *pile* est non-vide **faire**
4. dépiler t le sommet de *pile*
5. **pour** chaque état u avec un arc de t à u étiqueté par ϵ, **faire**
6. ajouter u à $\epsilon\!-\!fermeture(T)$
7. empiler u sur *pile*
   fin
   fin

Algorithme de construction de l'AFD.
La méthode utilisée consiste à construire les états de l'AFD comme des ensembles d'états de l'AFN, et de construire la fonction de transition entre ces états de telle façon qu'elle simule en parallèle toutes les transitions possibles dans l'AFN. On va donc construire *DÉtats* et *DTrans*, respectivement l'ensemble des états et la fonction de transition de l'AFD. Soit e_0 l'état initial de l'AFN, l'algorithme de calcul de $\epsilon\!-\!fermeture(T)$ et la fonction $Transiter(T, a)$ qui retourne tous les états de l'AFN vers lesquels il existe une transition sur a depuis un état de T :

1. Au départ, *DÉtats* contient un seul état initial non-marqué $\epsilon\!-\!fermeture(e_0)$
2. **tant que** il existe un état non-marqué T dans *DÉtats* **faire**
3. marquer T
4. **pour** chaque symbole d'entrée $a \in \Sigma$ **faire**
5. $U := \epsilon\!-\!fermeture(Transiter(T, a))$
6. **si** U n'appartient pas à *DÉtats* **alors**
7. ajouter U comme état non-marqué à DÉtats
8. $DTran[T, a] := U$
   fin
   fin

FIG. 2.4 – Construction d'un AFD équivalent à un AFN.

Corollaire 2.1 *(équivalence AFN/ER) Soit L un langage accepté par un AFN A, alors il existe une expression régulière e telle que $L(A) = L(e)$.*

Preuve. Découle directement du théorème d'équivalence AFD/AFN et du théorème d'équivalence AFD/ER. *CQFD.*

L'équivalence entre expressions régulières et automates finis déterministes est extrêmement importante en pratique, car cela veut dire que les langages définis par les expressions régulières peuvent être acceptés par des programmes consommant peu de ressources : une quantité finie de mémoire pour un langage donné. Nous allons voir au prochain chapitre à quel point les langages réguliers peuvent nous rendre de précieux services.

L'importance de la seconde équivalence, entre AFD et AFN, est moins immédiatement visible. En fait, il est plus simple de créer un automate fini non-determinste par la méthode de Thompson que de créer directement un automate fini déterministe. Certains logiciels choisissent donc de passer par un automate non-déterministe. Le défaut cependant du passage par un automate fini non-déterministe est la taille de l'automate fini déterministe produit. En effet, les états de l'AFD produit par l'algorithme de la figure 2.4 étant des sous-ensembles des états de l'AFN, le nombre potentiel d'états de l'AFD peut aller jusqu'au nombre de sous-ensembles de l'ensemble des états de l'AFN, qui est exponentiel puisque le nombre de sous-ensembles d'un ensemble de taille n est 2^n.

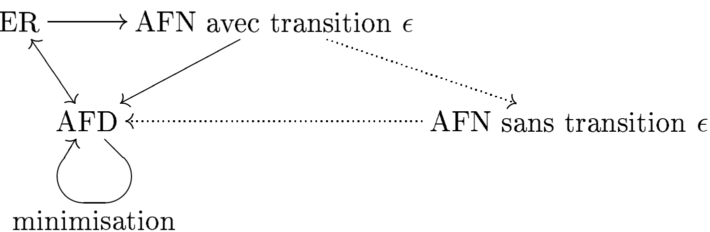
Il n'y a cependant pas d'unicité d'automate fini déterministe pour accepter un certain langage L. Au contraire, plusieurs automates peuvent reconnaître un même langage. Une caractéristique importante des automates est bien sûr leur taille en nombre d'états. Il se trouve que parmi tous les automates qui peuvent reconnaître un même langage, il existe un automate minimal unique au renommage de ses états près.

Théorème 2.3 *(AFD minimal) parmi tous les AFD reconnaissant un même langage L, il en existe un (à un renommage de ses états près) possédant le plus petit nombre d'états nécessaire pour reconnaître L.*

Preuve. Encore une fois, la preuve de ce théorème est obtenue par construction. L'idée de l'algorithme présenté à la figure 2.5 consiste à construire l'automate minimal par des états qui sont des classes d'équivalences sur les états de l'AFD à minimiser. Pour trouver ces classes, on commence par une partition minimale des états : les états finaux d'un côté et les non-finaux de l'autre. À chaque itération, l'algorithme divise les parties de la partition courante à chaque fois qu'un groupe d'états de cette partie ne se comporte pas comme les autres états de la partie sur un ou plusieurs symboles d'entrées en transitant vers un état différent des autres. L'automate obtenu est minimal, ce qui est démontrable par contradiction. Si un automate ayant moins d'état que l'AFD que nous venons de construire pouvait reconnaître le même langage que ce dernier, alors on devrait pouvoir l'obtenir en fusionnant certains états de l'automate obtenu. Or cette fusion n'est pas possible puisque les états en question n'auraient pas le même comportement transitoire. *CQFD.*

Le théorème de minimisation des automates d'états finis rend praticable la construction d'un automate déterministe en passant par un automate non-déterministe obtenu par la construction de Thompson. Bien que la construction de l'AFD intermédiaire puisse donner un automate de très grande taille, sa minimisation nous assure de toutes façons de trouver le plus petit automate reconnaissant le langage. Certes, il faudra fournir un effort pour le minimiser, mais cela est fait une fois pour toutes.

Le petit schéma suivant résume les constructions que nous avons présentées dans cette section permettant de passer d'une expression régulière à des automates et vice versa, puis d'automates entre eux (sauf le passage aux AFN sans ϵ-transition que nous n'avons pas explicitement présenté) :



Minimisation d'un AFD.

À partir d'un AFD quelconque $M = (Q, \Sigma, \delta, q_0, F)$, contruire un AFD M' tel que $L(M) = L(M')$ et tel que M' soit le plus petit AFD reconnaissant $L(M)$ (celui qui contient le moins d'états).

1. soit Π une partition initiale de Q contenant deux ensembles : F et $Q \setminus F$.
2. initialiser Π_n à la partition vide.
3. **tant que** $\Pi \neq \Pi_n$ **faire**
4. **pour** chaque ensemble d'états G de Π **faire**
5. partitionner G en sous-ensembles de telle façon que deux états e et t de G soient dans le même sous-ensemble ssi $\forall a \in \Sigma$ les états e et t ont des transitions sur a vers des états du même ensemble dans Π.
6. remplacer G dans Π_n par tous les sous-ensembles ainsi formés.

   fin
7. interchanger Π et Π_n.

   fin
8. Construire M' en prenant comme états q'_i les ensemble d'états de M calculés dans Π ; dans la fonction δ', il y aura une transition de q'_i vers q'_j ssi il existait dans δ une transition d'un état $q_m \in q'_i$ vers un état $q_n \in q'_j$; l'état initial de M' est l'état q'_0 contenant q_0 et les états finaux sont ceux obtenus de la partition de F.
9. Retirer de M' tout état non-acceptant qui n'a de transitions que vers lui-même ou tout état inatteignable depuis l'état initial.

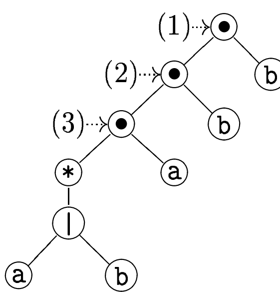
FIG. 2.5 – Minimisation d'un automate fini déterministe.

2.4 Mise en pratique des algorithmes

Dans cette section, nous allons voir successivement comment construire un AFN à partir d'une expression régulière grâce à la construction de Thompson, puis comment transformer cet AFN en AFD et ensuite le minimiser. Nous verrons ensuite comment construire un AFD directement à partir de l'expression régulière.

2.4.1 Construction de Thompson

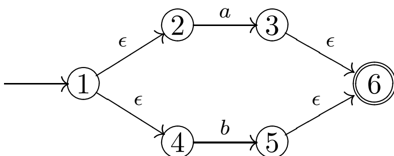
Considérons l'expression régulière `(a|b)*abb`. La première étape (voir 2.2) consiste à décomposer l'expression régulière en ses sous-expressions de manière à appliquer la construction de Thompson par induction sur la structure de l'expression. Dans le cas de l'expression précédente, voici l'arborescence que nous obtenons (les étiquettes (1), (2) et (3) nous serviront de référence par la suite) :



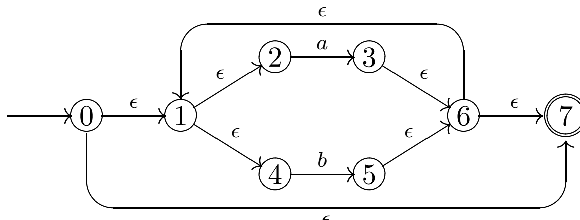
La construction se fait par induction sur la structure de l'expression régulière. Considérons d'abord la sous-expression a la plus à gauche (en bas de l'arbre). Par la règle 2 de la construction de Thompson, on obtient :

De même, pour la sous-expression b juste à droite de la précédente, on obtient par la même règle :

La sous-expression d'alternative prenant les deux sous-expressions précédentes en argument donne lieu à la construction suivante par la règle 3a :

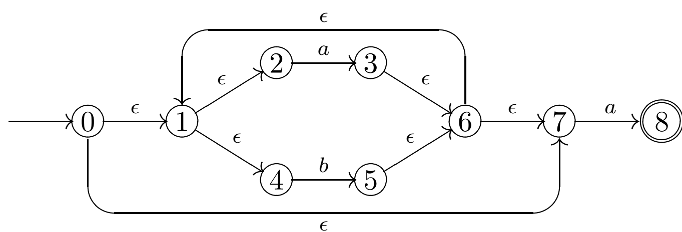


Maintenant, il faut appliquer l'opérateur d'itération '*' sur l'expression précédente. Par la règle 3c, on doit ajouter un état initial 0 et un état final 7. Une ϵ-transition relie l'état 0 à l'état 1 pour reconnaître l'expression précédente une fois ou plus. Une ϵ-transition relie aussi l'état 0 à l'état final 7 pour ne reconnaître que la chaîne vide. Une autre ϵ-transition relie l'état 6 à l'état 7 pour finir la reconnaissance après une ou plusieurs reconnaissances de l'expression précédente. Enfin, une ϵ-transition relie l'état 6 à l'état 1 pour répéter la reconnaissance de l'expression précédente autant de fois que l'on veut. Ceci donne l'AFN :

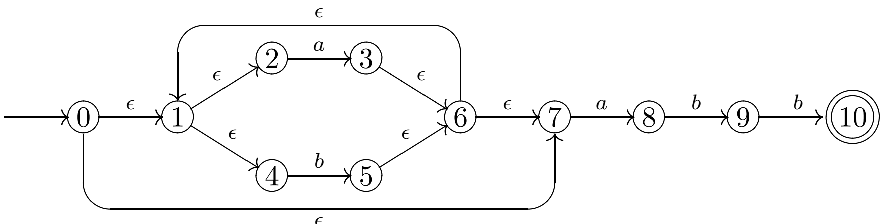


En appliquant à nouveau la règle 2 sur la sous-expression a suivant immédiatement l'opérateur de fermeture, on trouve à nouveau un petit AFN comme les précédents :

Maintenant, il faut s'attaquer à l'opérateur de concaténation étiqueté (3) dans l'arbre et qui relie les deux sous-expressions précédentes. On applique alors la règle 3b en superposant les états 7 et 7' des deux précédents AFN et en faisant de l'état 8 l'état final du nouvel AFN :



On répète ensuite des opérations similaires pour traiter les deux dernières sous-expressions `b` et leurs concaténations successives à l'automate résultant, ce qui nous donne comme résultat final l'AFN suivant :



2.4.2 Construction d'un AFD à partir d'un AFN

L'algorithme de construction d'un AFD à partir d'un AFN (voir 2.4) consiste à simuler en parallèle toutes les transitions possibles à partir d'un certain état sur un symbole d'alphabet donné. Il faut également regrouper en un état tous les états accessibles par ϵ-transition à partir de chacun des états construits, d'où l'utilisation d'une procédure d'ϵ-fermeture qui consiste tout simplement à trouver pour un ensemble d'états d'un AFN tous les états accessibles par ϵ-transitions.

Considérons l'AFN obtenu à la section précédente pour l'expression régulière `(a|b)*abb` et déroulons les étapes de l'itération de la procédure de construction de l'AFD.

Étape 1

DÉtats $\leftarrow$ $\{\epsilon\text{-fermeture}(\{0\})\}$

Calculons au long cette ϵ-fermeture. Initialement la pile contient l'état 0 ; notons cela pile $\leftarrow$ [0]. Le résultats courant est $\{0\}$. À la première itération, on considére l'état 0 en sommet de pile. Les états accessibles par ϵ-transition à partir de 0 sont 1 et 7. On les empile et on les ajoute au résultat, ce qui nous donne pile $\leftarrow$ [1,7] et résultat $\leftarrow$ $\{0,1,7\}$.

À la seconde itération, on considère l'état 7 qui apparaît en sommet de pile ; aucun état étant accessible par ϵ-transition à partir de 7, la pile ne contient plus que l'état 1 et le résultat demeure inchangé.

À la troisième itération, on considère l'état 1. Les états accessibles par ϵ-transition à partir de 1 sont 2 et 4. On les empile et on les ajoute au résultat, ce qui donne pile $\leftarrow$ [2,4] et résultat $\leftarrow$ $\{0,1,2,4,7\}$.

À la quatrième itération, on considère l'état 4 ; aucun état étant accessible par ϵ-transition à partir de 4, la pile ne contient plus que l'état 2 et le résultat demeure inchangé.

À la cinquième itération, on considère l'état 2 ; aucun état étant accessible par ϵ-transition à partir de 2, la pile devient vide et le résultat demeure inchangé.

L'algorithme d'ϵ-fermeture s'arrête alors puisque la pile est vide. L'état initial de l'AFD est donc construit comme l'ensemble d'états {0,1,2,4,7} de l'AFN.

Étape 2

DÉtats $\leftarrow$ $\{\{0,1,2,4,7\}\}$

Le seul état actuellement dans l'ensemble d'états DÉtats est sélectionné et on examine ses transitions possibles pour construire la fonction de transition de l'AFD appelée DTrans. Ici, il existe uniquement deux symboles d'entrée : `a` et `b`.

$$\begin{aligned} DTrans[\{0,1,2,4,7\},\mathtt{a}] &= \epsilon\text{-fermeture}(NTrans[0,\mathtt{a}] \cup NTrans[1,\mathtt{a}] \cup NTrans[2,\mathtt{a}] \cup \\ &\quad NTrans[4,\mathtt{a}] \cup NTrans[7,\mathtt{a}]) \\ &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{3\} \cup \{\} \cup \{8\}) \\ &= \epsilon\text{-fermeture}(\{3,8\}) \\ &= \{1,2,3,4,6,7,8\} \end{aligned}$$

Cet état n'existant pas encore, on le rajoute à DÉtats. Considérons ensuite les transitions sur `b` :

$$\begin{aligned} DTrans[\{0,1,2,4,7\},\mathtt{b}] &= \epsilon\text{-fermeture}(NTrans[0,\mathtt{b}] \cup NTrans[1,\mathtt{b}] \cup NTrans[2,\mathtt{b}] \cup \\ &\quad NTrans[4,\mathtt{b}] \cup NTrans[7,\mathtt{b}]) \\ &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{\} \cup \{5\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{5\}) \\ &= \{1,2,4,5,6,7\} \end{aligned}$$

Cet état n'existant pas non plus, on le rajoute à DÉtats.

Étape 3

DÉtats $\leftarrow$ $\{\overline{\{0,1,2,4,7\}},\{1,2,3,4,6,7,8\},\{1,2,4,5,6,7\}\}$

On sélectionne l'état {1,2,3,4,6,7,8}. Les transitions sur `a` sont :

$$\begin{aligned} DTrans[\{1,2,3,4,6,7,8\},\mathtt{a}] &= \epsilon\text{-fermeture}(\{\} \cup \{3\} \cup \{\} \cup \{\} \cup \{\} \cup \{8\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{3,8\}) \\ &= \{1,2,3,4,6,7,8\} \end{aligned}$$

Cet état existant déjà, on n'a pas à le rajouter à DÉtats. Considérons les transitions sur `b` :

$$\begin{aligned} DTrans[\{1,2,3,4,6,7,8\},\mathtt{b}] &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{\} \cup \{5\} \cup \{\} \cup \{\} \cup \{9\}) \\ &= \epsilon\text{-fermeture}(\{5,9\}) \\ &= \{1,2,4,5,6,7,9\} \end{aligned}$$

Cet état n'existant pas déjà, on le rajoute à DÉtats.

Étape 4

DÉtats $\leftarrow$ $\{\overline{\{0,1,2,4,7\}}, \overline{\{1,2,3,4,6,7,8\}}, \{1,2,4,5,6,7\}, \{1,2,4,5,6,7,9\}\}$

On sélectionne l'état {1,2,4,5,6,7} et on examine ses transitions :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7\},\texttt{a}] &= \epsilon\text{-fermeture}(\{\} \cup \{3\} \cup \{\} \cup \{\} \cup \{\} \cup \{8\}) \\ &= \epsilon\text{-fermeture}(\{3,8\}) \\ &= \{1,2,3,4,6,7,8\} \end{aligned}$$

Cet état existant déjà, on n'a pas à le rajouter à DÉtats. Considérons les transitions sur `b` :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7\},\texttt{b}] &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{5\} \cup \{\} \cup \{\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{5\}) \\ &= \{1,2,4,5,6,7\} \end{aligned}$$

Cet état existant déjà, on n'a pas à le rajouter à DÉtats.

Étape 4

DÉtats $\leftarrow$ $\{\overline{\{0,1,2,4,7\}}, \overline{\{1,2,3,4,6,7,8\}}, \overline{\{1,2,4,5,6,7\}}, \{1,2,4,5,6,7,9\}\}$

On sélectionne l'état {1,2,4,5,6,7,9}. Ses transitions sur `a` sont :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7,9\},\texttt{a}] &= \epsilon\text{-fermeture}(\{\} \cup \{3\} \cup \{\} \cup \{\} \cup \{\} \cup \{8\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{3,8\}) \\ &= \{1,2,3,4,6,7,8\} \end{aligned}$$

Cet état existant déjà, on n'a pas à le rajouter à DÉtats. Considérons les transitions sur `b` :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7,9\},\texttt{b}] &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{5\} \cup \{\} \cup \{\} \cup \{\} \cup \{10\}) \\ &= \epsilon\text{-fermeture}(\{5,10\}) \\ &= \{1,2,4,5,6,7,10\} \end{aligned}$$

Cet état n'existant pas déjà, on le rajoute à DÉtats.

Étape 5

DÉtats $\leftarrow$ $\{\overline{\{0,1,2,4,7\}}, \overline{\{1,2,3,4,6,7,8\}}, \overline{\{1,2,4,5,6,7\}}, \overline{\{1,2,4,5,6,7,9\}}, \{1,2,4,5,6,7,10\}\}$

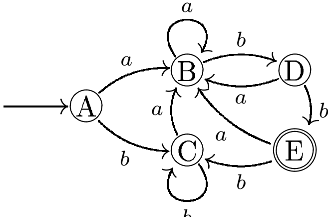
On sélectionne l'état {1,2,4,5,6,7,10}. Ses transitions sur `a` sont :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7,10\},\texttt{a}] &= \epsilon\text{-fermeture}(\{\} \cup \{3\} \cup \{\} \cup \{\} \cup \{\} \cup \{8\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{3,8\}) \\ &= \{1,2,3,4,6,7,8\} \end{aligned}$$

Cet état existant déjà, on n'a pas à le rajouter à DÉtats. Considérons les transitions sur `b` :

$$\begin{aligned} DTrans[\{1,2,4,5,6,7,10\},\mathtt{b}] &= \epsilon\text{-fermeture}(\{\} \cup \{\} \cup \{5\} \cup \{\} \cup \{\} \cup \{\} \cup \{\}) \\ &= \epsilon\text{-fermeture}(\{5\}) \\ &= \{1,2,4,5,6,7\} \end{aligned}$$

Cet état existe déjà aussi, et donc DÉtats demeure inchangé. Les états de DÉtats étant maintenant tous marqués, donc traités, l'algorithme s'arrête. Si on représente de manière graphique l'AFD résultant, on obtient :



où A = {0,1,2,4,7}, B = {1,2,3,4,6,7,8}, C = {1,2,4,5,6,7}, D = {1,2,4,5,6,7,9}, et E {1,2,4,5,6,7,10}.

2.4.3 Construction directe de l'AFD à partir de l'expression régulière

L'algorithme de construction d'un AFD à partir de l'expression régulière (voir 2.3) est moins évident encore que le précédent. L'idée générale est la suivante. On numérote les positions dans l'expression régulière où apparaissent les symboles terminaux. Les états vont en quelque sorte représenter de ensembles de positions atteintes depuis le début de la reconnaissance de la chaîne, et les transitions vont se faire vers les positions suivantes atteintes si le prochain symbole dans la chaîne est reconnu par l'une des positions de l'état courant de l'automate.

La fonction capitale à obtenir est donc la fonction *PosSuivante* qui, pour chaque position dans l'expression régulière, retourne les positions suivantes qui peuvent être atteintes. Pour calculer cette fonction, on passe par trois autres fonctions : *Annulable*, *PremièrePos* et *DernièrePos*. Ces fonctions s'appliquent sur les nœuds de l'arbre de l'expression régulière et calculent respectivement si oui ou non un nœud peut reconnaître la sous-chaîne vide, l'ensemble des premières positions auxquelles on peut reconnaître le prochain symbole dans la chaîne et l'ensemble des dernières positions auxquelles on peut reconnaître le prochain symbole dans la chaîne.

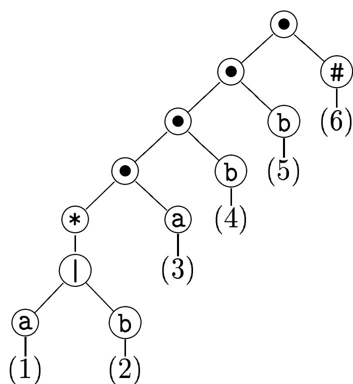
Muni des trois fonctions *Annulable*, *PremièrePos* et *DernièrePos*, on calcule la fonction *PosSuivante*, à partir de laquelle on construit l'AFD. Considérons encore l'expression régulière `(a|b)*abb` pour illustrer cet algorithme.

Première étape : numérotation des positions

On ajoute à l'expression régulière le symbole '`#`' qui sert de marqueur de fin de l'expression, puis on construit l'arbre de l'expression où on numérote les positions correspondant aux symboles de l'alphabet d'entrée augmenté de '`#`' :

Nœuds	*Annulable*	*PremièrePos*	*DernièrePos*
(1)	faux	{1}	{1}
(2)	faux	{2}	{2}
(3)	faux	{3}	{3}
(4)	faux	{4}	{4}
(5)	faux	{5}	{5}
(6)	faux	{6}	{6}
| (a, b)	$A(1)$ ou $A(1)$ = faux	$P(1) \cup P(2) = \{1{,}2\}$	$D(1) \cup D(2) = \{1{,}2\}$
* (|)	vrai	$P(\|) = \{1{,}2\}$	$D(\|) = \{1{,}2\}$
• (*, a)	faux	$P(*) = \{1{,}2\}$	$D(\mathtt{a}) = \{3\}$
• (•, b)	faux	$P(\bullet) = \{1{,}2\}$	$D(\mathtt{b}) = \{4\}$
• (•, b)	faux	$P(\bullet) = \{1{,}2\}$	$D(\mathtt{b}) = \{5\}$
• (•, #)	faux	$P(\bullet) = \{1{,}2\}$	$D(\#) = \{6\}$

FIG. 2.6 – Tableau des fonctions *Annulable*, *PremièrePos* et *DernièrePos* pour l'expression régulière `(a|b)*abb`.



Calcul des fonctions *Annulable*, *PremièrePos* et *DernièrePos*

On fabrique un tableau dont les entrées horizontales sont les nœuds de l'arbre de l'expression régulière et les entrées verticales sont les trois fonctions. Le tableau obtenu apparaît à la figure 2.6. À partir de ces trois fonctions, on calcule la fonction *PosSuivante*. Contrairement aux trois précédentes, la fonction *PosSuivante* prend en entrée des positions, et non des nœuds, et elle retourne des ensembles de positions. Il suffit donc de la calculer pour les positions 1 à 6.

Pour la position 1, on constate que 1 apparaît dans *DernièrePos*(∗), donc par la règle des itérations on ajoute *PremièrePos*(∗). Par ailleurs, le nœud de concaténation s'appliquant sur

l'itération a 1 dans les dernières positions de son nœud de gauche, donc on ajoute les premières positions de son nœud de droite, c'est-à-dire *PremièrePos*(a) = {3}. On obtient donc :

$PosSuivante(1) = \{1,2\} \cup \{3\} = \{1,2,3\}$

Pour la position 2, on a exactement le même raisonnement que pour la position 1, ce qui nous donne :

$PosSuivante(2) = \{1,2\} \cup \{3\} = \{1,2,3\}$

Pour la position 3, cette position apparaît dans les dernières positions du nœud de concaténation qui a pour nœud de droite le b en position 4. On a donc :

$PosSuivante(3) = \{4\}$

Pour les positions 4 et 5, encore une fois elles apparaissent dans les dernières positions du nœud de gauche d'un nœud de concaténation dont les premières positions respectives du nœud de droite sont {5} et {6}, d'où :

$PosSuivante(4) = \{5\}$

$PosSuivante(5) = \{6\}$

Pour la position 6, aucune règle s'applique ; l'ensemble de ses positions suivantes est donc vide.

Construction de l'AFD

Un peu à la manière de l'algorithme de construction de l'AFD à partir de l'AFN, le présent algorithme construit l'AFD à partir d'un état initial et en examinant les transitions à partir de chaque état construit. L'état initial est obtenu par les premières positions du nœud racine de l'arbre de l'expression régulière :

DÉtats = $\{\{1,2,3\}\}$

Première itération

On sélection l'état {1,2,3} pour calculer les états obtenu par transition sur chacun des symboles de l'alphabet d'entrée Σ = {a,b}. Pour savoir quelles sont les positions à ajouter à l'état vers lequel on va, il faut cumuler les positions suivantes de toutes les positions dans l'état courant qui correspondent au symbole d'entrée. Pour le symbole 'a', les positions de l'état courant sont 1 et 3. on a donc :

DTrans[{1,2,3},a] = *PosSuivante*(1) ∪ *PosSuivante*(3) = {1,2,3} ∪ {4} = {1,2,3,4}

Cet état n'existant pas, on l'ajoute à DÉtats.

Pour le symbole 'b', la seule position qui corresponde est 2 ; on prend donc ses positions suivantes :

DTrans[{1,2,3},b] = *PosSuivante*(2) = {1,2,3}

Cet état est l'état courant, donc on ne l'ajoute pas à DÉtats.

Deuxième itération

DÉtats = $\{\overline{\{1,2,3\}}, \{1,2,3,4\}\}$

On sélectionne l'état {1,2,3,4}, parmi lequel les positions correspondant à un 'a' sont 1 et 3 et celles correspondant à un 'b' sont 2 et 4. On a donc :

DTrans[{1,2,3,4},a] = *PosSuivante*(1) ∪ *PosSuivante*(3) = {1,2,3,4}

DTrans[{1,2,3,4},b] = $PosSuivante(2) \cup PosSuivante(4) = \{1,2,3\} \cup \{5\} = \{1,2,3,5\}$

Ce dernier état est ajouté à DÉtats.

Troisième itération

DÉtats = $\{\overline{\{1,2,3\}}, \overline{\{1,2,3,4\}}, \{1,2,3,5\}\}$

On sélectionne l'état {1,2,3,5}, parmi lequel les positions correspondant à un 'a' sont encore 1 et 3 et celles correspondant à un 'b' sont 2 et 5. On a donc :

DTrans[{1,2,3,5},a] = $PosSuivante(1) \cup PosSuivante(3) = \{1,2,3,4\}$

DTrans[{1,2,3,5},b] = $PosSuivante(2) \cup PosSuivante(5) = \{1,2,3\} \cup \{6\} = \{1,2,3,6\}$

Ce dernier état est ajouté à DÉtats.

Quatrième itération

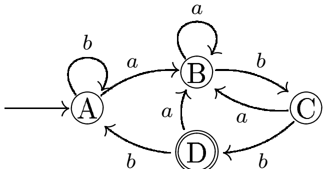
DÉtats = $\{\overline{\{1,2,3\}}, \overline{\{1,2,3,4\}}, \overline{\{1,2,3,5\}}, \{1,2,3,6\}\}$

On sélectionne l'état {1,2,3,6}, parmi lequel les positions correspondant à un 'a' sont encore 1 et 3 et celle correspondant à un 'b' est 2. On a donc :

DTrans[{1,2,3,6},a] = $PosSuivante(1) \cup PosSuivante(3) = \{1,2,3,4\}$

DTrans[{1,2,3,6},b] = $PosSuivante(2) = \{1,2,3\}$

Ces états existent déjà et donc ne sont pas ajoutés à DÉtats. Tous les états de DÉtats sont maintenant marqués, ce qui termine l'algorithme. Si on note bien que les états finaux de l'AFD sont ceux qui contiennent la position du symbole # ajouté à l'expression régulière, l'automate résultant est :

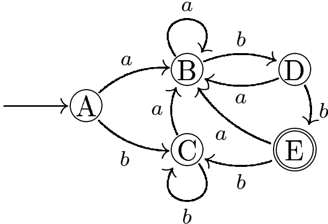


où A = {1,2,3}, B = {1,2,3,4}, C = {1,2,3,5}, et D = {1,2,3,6}.

2.4.4 Minimisation d'un AFD

L'algorithme de minimisation se base sur le principe de regroupement en classes d'équivalence des états de l'automate initial qui ont le même comportement. Même comportement dans ce cas veut dire les mêmes transitions sur les symboles d'entrée. Le principe général a déjà été évoqué. On va partitionner les états de l'AFD de manière à chercher les classes d'équivalence d'états.

Considérons l'automate obtenu de l'expression régulière `(a|b)*abb` par la construction de Thompson puis transformation en AFD :



La première étape détermine une partition minimale de l'AFD : d'un côté les états finaux

et de l'autre les non-finaux. C'est en effet sur le comportement d'acceptation ou de non-acceptation que l'on doit nécessairement distinguer les états. Pour notre automate, cela donne comme première partition Π_0 :

$\Pi_0 = \{Q \backslash F, F\} = \{\{A, B, C, D\}, \{E\}\}$

Maintenant, il faut vérifier chacun des sous-ensembles d'états pour voir si tous les états qui appartiennent à ce sous-ensemble ont bien le même comportement sur leurs transitions. Il s'agit donc de vérifier si pour tous les états du sous-ensemble s'ils transitent tous vers des états du même sous-ensemble de la partition sur chacun des symboles d'entrée. Dans la partition précédente, les états A, B et C transitent vers le même sous-ensemble à la fois pour a et b, mais l'état D, sur a, transite lui vers l'état E, qui est dans l'autre sous-ensemble.

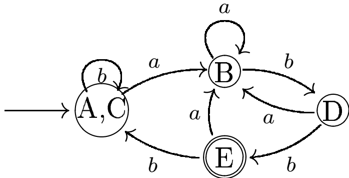
On va donc passer à une nouvelle partition où l'état D est séparé du premier sous-ensemble. L'un des points à vérifier avec précaution lors de cette étape est de créer le minimum de nouveaux sous-ensembles. Ainsi, il faut maintenir dans le même sous-ensemble tous les états qui ont le même comportement vis-à-vis de la partition courante. Ici, nous n'avons pas ce problème et on obtient une deuxième partition :

$\Pi_1 = \{\{A, B, C\}, \{D\}, \{E\}\}$

On reprend alors une nouvelle phase de vérification. Ici, A et C vont vers B sur a et vers A ou C sur b ; ils ont donc le même comportement. Par contre, B transite bien vers le même sous-ensemble sur a mais pas sur b où il transite vers D. On doit donc séparer B du premier sous-ensemble, ce qui donne une nouvelle partition :

$\Pi_2 = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$

À cette étape, en vérifiant, on se rend compte que tous les états ont le même comportement de transition. La partition résiste à la vérification, ce qui arrête l'algorithme. L'AFD minimal obtenu est construit à partir de la partition, sachant qu'est état final tout sous-ensemble contenant un état final de l'automate initial.



Notons ici que cet automate est le même, à un renommage de ses états près, que celui obtenu par la construction directe, ce qui nous indique que l'algorithme de construction directe donne de fait l'automate minimal.

2.5 Exercices

2.5.1. Construire l'AFN reconnaissant le langage défini par l'expression régulière `(ba*|ab)bb`, en utilisant la construction de Thompson.

2.5.2. Construire un AFD à partir de l'AFN produit à l'exercice précédent.

2.5.3. Construire un AFD directement à partir de l'expression régulière `(ba*|ab)bb`.

2.5.4. Minimiser l'AFD obtenu pour `(ba*|ab)bb` en passant par l'AFN.

Chapitre 3

Application des expressions régulières

Malgré leur possibilités certes limitées, la puissance des expressions régulières en font des auxiliaires très intéressants en informatique. De fait, de nombreux outils utilisent les expressions régulières : recherche et remplacement dans les éditeurs de texte, commandes Unix, langages de scripts (Perl, Python, ...), librairie d'expressions régulières des langages généralistes comme Java, etc. Ce chapitre vise à donner une première introduction à cette utilisation des expressions régulières dans les outils courants.

3.1 L'outil egrep

L'outil `egrep` est l'un des outils les plus simples à utiliser les expressions régulières. Il s'agit d'une commande popularisée sous Unix mais maintenant disponibles sous de nombreux systèmes d'exploitation. À la base, on lui fournit en entrée une expression régulière et une liste de fichiers, et `egrep` retourne en résultat toutes les lignes des fichiers d'entrée où il a été possible d'apparier quelque chose avec l'expression régulière. Par exemple, si on fait :

```
> egrep 'abc' monFichier.txt
```

la commande va traiter le fichier `monFichier.txt` ligne par ligne, essayer d'apparier l'expression régulière `abc`[1] et retourner toutes les lignes où cela aura été possible. En Unix, cette commande s'intègre parfaitement dans l'esprit selon lequel les commandes prennent en entrée des fichiers séquentiels de caractères et retournent des fichiers séquentiels de caractères. Ceci permet donc de pipeliner les commandes `egrep` avec d'autres commandes Unix, comme par exemple la commande `wc` (pour `word count` bien sûr). La commande composée suivante :

```
> egrep 'abc' monFichier.txt | wc
```

retourne trois nombres (résultant de `wc`) : le nombre de mots, de lignes et de caractères dans le fichier résultant de la commande `egrep`

L'outil `egrep`, comme la plupart des outils qui utilisent les expressions régulières, ne se limite pas aux expressions de la théorie telles que nous les avons vues au chapitre précédent. Bien que ces expressions régulières «canoniques» soient suffisantes pour exprimer tous les langages réguliers, il est souvent pratique d'utiliser des raccoucis. Par exemple, si on veut exprimer le fait qu'on veut apparier une certaine expression régulière e ou la chaîne vide, on

[1] Ici, l'expression régulière est mise entre apostrophe pour éviter l'interprétation de ses caractères par l'interpréteur de commandes, ce qui est souvent nécessaires lorsqu'on utilise des métacaractères comme nous le verrons plus loin.

Métacaractère	Apparie
`.`	n'importe quel caractère sauf la fin de ligne.
`[ ]`	(classe de caractères) tout caractère dans la séquence entre crochet.
`[^ ]`	(classe de caractères complémentaire) tout caractère sauf ceux dans la séquence.
c	échappement de métacaractère ; si *c* est un métacaractère, il reprend sa signification normale en tant que caractère.
`^`	la position au début de la ligne.
`$`	la position à la fin de la ligne.
`\<`	la position au début d'un mot.
`\>`	la position à la fin d'un mot.
`|`	(alternative) ce qu'apparie la sous-expression de gauche ou la sous-expression de droite.
`?`	(option) 0 ou une fois l'appariement de la sous-expression.
`*`	(itération, fermeture) 0, une ou plusieurs fois l'appariement de la sous-expression.
`+`	une ou plusieurs fois l'appariement de la sous-expression.
{*min*, *max*}	entre *min* fois et *max* fois l'appariement de la sous-expression.
`( )`	groupement limitant la portée et utilisé pour les références arrières.
i	une nouvelle fois ce qui a été apparié par le $i^{\text{ième}}$ groupe.

FIG. 3.1 – Principaux métacaractères des expressions régulières de egrep

peut exprimer cela de la manière suivante :

$(e|\epsilon)$

Ce genre d'expression qui cherche simplement à exprimer le fait que l'appariement de e est optionnel, devient vite assez lourd à écrire et à lire. `egrep` propose plutôt l'opérateur '`?`' pour signifier l'optionnalité, ce qui nous permettrait d'écrire ici :

e`?`

De même, si on veut écrire qu'à un certain endroit on veut apparier toutes les minuscules, il faudrait écrire dans la version canonique :

`(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)`

ce qui est bien sûr long et fastidieux. `egrep` permet d'écrire des classes de caractères sous la forme d'une séquence de caractères entre crochets. On peut donc écrire les lettres minuscules sous la forme :

`[abcdefghijklmnopqrstuvwxyz]`

Cette forme va apparier exactement un caractère parmi la liste. Elle est déjà un peu plus compacte mais encore fastidieuse. Profitant du fait que l'alphabet est représenté en ASCII et en latin-1 par des codes numériques séquentiels, `egrep` permet d'écrire des intervalles de caractères en spécifiant le premier et le dernier caractère dans l'ordre numérique des codes, séparés par le tiret '-'. On peut donc tout simplement écrire :

`[a-z]`

ce qui est nettement plus confortable. On peut avoir plusieurs intervalles dans une classe de

caractères. Ainsi,

```
[a-zA-Z]
```

va apparier exactement un caractère parmi les minuscules et les majuscules.

Ici, le caractère tiret est pris avec un sens particulier, ce qui semble empêcher de l'inclure dans une classe de caractère avec son sens original, c'est-à-dire apparier le caractère tiret. De fait, il existe un moyen de contourner cela : si on veut avoir une classe de caractère qui apparie aussi le caractère tiret, il faut placer le caractère tiret au début de la classe, comme par exemple dans :

```
[-_]
```

qui va apparier le tiret ou le souligné.

Il existe aussi une autre forme intéressante des classes de caractères : la classe de caractères complémentaire. Quand une classe de caractères commence par le caractère caret '^', alors elle apparie non pas les caractères dans la liste mais tout caractère du jeu de caractère sauf ceux qui apparaissent dans la liste. La classe

```
[^a-zA-Z]
```

apparie donc tout caractère sauf les lettres majuscules et minuscules. `egrep` définit plusieurs classes de caractères selon une syntaxe `[[:`*<nom>* `:]]` ; consultez la page de manuel de `egrep` pour les connaître. Il y a aussi un autre métacaractère très utile : le métacaractère '`.`' (le point) qui signifie apparier tout caractère sauf la fin de ligne.

La figure 3.1 résume les principaux métacaractères de `egrep`. Vous êtes invités à consulter la page de manuel de `egrep` pour obtenir plus d'informations à ce sujet (faire '`man egrep`' sous Unix). Un concept intéressant parmi ces métacaractères est celui de positions appariées par des métacaractères. Le métacaratère '^', lorqu'il est utilisé en dehors d'une classe de caractères, signifie en fait qu'il faut apparier le début d'une ligne. Par exemple,

```
^From:
```

n'appariera la séquence de caractères '`From:`' que si celle-ci apparaît au début d'une ligne. Les métacaractères '`$`', '`\<`' et '`\>`' jouent le même rôle pour respectivement la fin de ligne, le début d'un mot et la fin d'un mot.

Expressions régulières augmentées

Si la plupart des ajouts aux expressions régulières classiques ne sont que du «*sucre syntaxique*», c'est-à-dire une syntaxe plus simple pour écrire quelque chose qui aurait pu s'écrire dans la syntaxe classique, `egrep` fournit une autre construction qui est très puissante et en fait en dehors des expressions régulières : le groupement par parenthèses et la référence arrière. Les parenthèses peuvent être libéralement utilisées pour délimiter la portée des opérateurs, comme par exemple dans :

```
(abc)?
```

qui signifie apparier optionnellement la séquence `abc`. Sans les parenthèses, l'opérateur '`?`' ne s'appliquerait qu'au caractère `c` (il a plus forte priorité que l'opérateur de concaténation).

Cependant, outre ce rôle de délimitation de la portée, les parenthèses en `egrep` sont utilisées pour le regroupement et les références arrières exprimées par la syntaxe '$\backslash i$' où i réfère $i^{\text{ème}}$ groupe reconnu depuis le début de l'expression régulière. Par exemple,

```
([a-z]).*\1
```

va apparier d'abord un caractère parmi les lettre miniscules, puis une répétition de 0, 1, ou plusieurs caractères quelconques sauf la fin de ligne, puis à nouveau le même caractère qui a été reconnu par le premier groupe parenthésé. Pour savoir à quel groupe un indice i se réfère, il faut compter de gauche à droite les parenthèses ouvrantes et trouver le groupe introduit par la $i^{\text{ème}}$ parenthèse ouvrante. Il est bien sûr possible d'imbriquer les parenthèses.

La référence arrière n'est pas exprimable dans les expressions régulières théoriques, dans la mesure où le texte reconnu dans un groupement doit être mémoriser. Contrairement à ce qui se passe dans la reconnaissance par expressions régulières théoriques, la quantité de mémoire nécessaire pour représenter le texte reconnu n'est pas bornable. Il n'est donc pas possible de réaliser la référence arrière dans un automate d'états fini. C'est parce que la reconnaissance est faite par un ordinateur de plein droit, et donc qu'il est possible d'avoir accès à toute la puissance de ce dernier, qu'il est possible en `egrep` de proposer ces références arrières et d'offrir ainsi une puissance strictement plus grande que celle des expressions régulières classiques.

À propos des références arrières dans les expressions «régulières», il peut être utile de méditer la phrase suivante trouvée dans la documentation des expressions régulières standard POSIX (sous Linux) :

> «*Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does* '`a\(\(b\)*\2\)*d`' *match* '`abbbd`'*?). Avoid using them.*»

3.2 Autres commandes et outils Unix

Plusieurs commandes et outils disponibles sous Unix et parfois sous d'autres systèmes d'exploitation utilisent les expressions régulières. Dans cette section, nous en parcourons quelques uns de plus dont l'utilité peut s'avérer quotidienne.

3.2.1 La commande `find`

La commande Unix `find` sert à la recherche de fichiers dans une hiérarchie de répertoires. Son synoptique indique que cette commande prend deux arguments : le premier est un ou plusieurs chemins indiquant les racines des répertoires à parcourir, et le second argument est une commande à appliquer (par défaut, s'il n'y a pas de commande à appliquer, `find` imprime simplement les noms des fichiers trouvés.

La commande `find` est très générale ; elle permet de fouiller des répertoires pour trouver des fichiers en fonction de leur nom, leur type, leur groupe d'appartenance, par leur taille, par leur date de dernière modification, etc. Consultez le manuel en ligne ('`man find`') pour avoir plus de détails sur cette commande. L'option `-regex` permet de rechercher des fichiers par appariement de leur nom avec une expression régulière. Un point délicat à surveiller est que le nom de fichier utilisé est le nom complet, incluant le chemin des répertoires contenant le fichier.

Les expressions régulières (sous la variante de Linux installée sur machine) suivent le norme POSIX (faire '`man 7 regex`').

3.2.2 La commande sed

L'outil **sed** est un éditeur de texte «hors ligne» qui permet d'écrire des scripts d'édition de fichiers puis de les exécuter sous l'interpréteur de commande ou encore comme processus Unix en arrière plan (donc sans interface avec l'utilisateur). C'est donc une commande très utile lorsqu'il s'agit de réaliser des éditions pointues mais répétitives sur un important groupe de fichiers (peut-être sélectionnés par une commande `find`).

L'éditeur `sed` est relativement compliqué à utiliser. De plus, on lui préfère de plus en plus des scripts écrits dans des langages de scripts comme Perl ou Python. Dans les grandes lignes, `sed` s'applique à un fichier en le découpant en tampons qui correspondent en gros à des lignes. Chaque tampon est chargé en mémoire, puis les commandes d'éditions données dans un script `sed` sont appliquées. Lorsque les commandes ont été appliquées, le tampon résultant est copié dans le fichier résultant, puis on passe au tampon suivant.

L'intérêt de `sed` dans le contexte des expressions régulières vient de ce que l'on peut écrire des commandes de recherche et remplacement où le texte recherché est apparié à une expression régulière. Cette fonctionnalité nous offre une première occasion de parler d'une nouvelle utilité des groupements dans les expressions régulières : la possibilité d'utiliser les portions de chaînes reconnues par chacun des groupes après l'appariement, et donc ici pour construire le texte de remplacement. Avec une syntaxe similaire à celle des références arrières (mais qui n'en sont pas puisqu'elles ne sont plus utilisées dans l'expression régulière) on peut référencer le texte reconnu dans un groupe. Par exemple, la commande suivante :

```
sed -e 's/\([0-9][0-9]*\)\.\([0-9]*\)/\1,\2/g' monFichier.txt
```

prendra le fichier `monFichier.txt` et convertira les nombres décimaux en notation américaine (avec le point décimal) en notation française (avec la virgule décimale).

3.2.3 L'éditeur emacs

`Emacs` est un éditeur de texte extensible écrit en Lisp qui doit sa grande popularité non seulement au fait qu'il fut l'un des premiers éditeurs plein écran multifenêtré[2] mais aussi au fait qu'il est ouvert, au sens où il est relativement facile de définir de nouveaux modes d'édition avec toute la puissance de Lisp. cela va du simple mode français qui permet de composer les caractères accentués non-directement disponibles au clavier jusqu'au mode `jde` d'édition de Java qui propose plusieurs des fonctionnalités que l'on retrouve dans les environnements de programmation évolués (menus adaptatifs permettant de retrouver rapidement un fichier ou une définition dans un fichier, insertion de patrons syntaxiques, compilation, exécution, etc.).

`Emacs` offre de nombreuses fonctions utilisant les expressions régulières, dont l'un des exemples emblématiques est la commande de recherche et remplacement interactive avec expression régulière appelée par :

```
M-x query-replace-regex
```

où `M` dans le jargon `emacs` est la touche méta généralement réalisée par la touche échappement (ou escape) sur le clavier.

La syntaxe des expression régulières d'`emacs` est malheureusement différente des syntaxes vues jusqu'ici. Regardez la documentation en ligne d'`emacs` pour avoir plus de détails. Cepen-

[2]Emacs existait en effet avant même les écrans graphiques, et il permettait déjà avec des écrans purement textuels comme le vt100 d'éditer un texte en plein écran et même à subdiviser cet écran en plusieurs fenêtres horizontalement et verticalement !

dant, comme pour la recherche et remplacement de `sed`, la partie recherche permet d'utiliser des expressions régulières et il est possible d'utiliser les références «arrières» dans la partie remplacement pour réaliser une édition contextuelle d'un document.

3.3 Les expressions régulières de Perl

Le langage Perl a révolutionné le monde des langages de scripts à la fin des années '80 et demeure aujourd'hui, avec Python que nous évoquerons à la section suivante, l'un des langages de scripts les plus populaires. Un langage de scripts est un langage dont le rôle principal consiste à manipuler des séquences de caractères, chaînes et le plus souvent des fichiers. Il ne s'agit pas ici de faire un cours de Perl en soi. Les lecteurs intéressés sont priés de se référer à la documentation de Perl (voir par exemple [Pera]) pour avoir une idée générale de ce langage. L'intérêt de parler de Perl ici réside essentiellement dans le fait que la syntaxe des expressions régulières de Perl est en voie de devenir un standard *de facto* qui percole sur l'ensemble des outils. La documentation de Perl présente aussi une description de sa syntaxe des expressions régulières [Perb].

Une grande partie des métacaractères que nous avons vus jusqu'à maintenant apparaissent exactement de la même façon dans les expressions régulières Perl. L'anti-oblique sert à protéger les métacaractères pour leur redonner leur signification en tant que simple caractère. Le caret ('`^`') désigne la position en début de ligne, alors que le dollar ('`$`') désigne la position en fin de ligne. le point ('`.`') apparie tout caractère sauf la fin de ligne. L'alternative est dénoté par la verticale ('`|`'), le regroupement par les parenthèses et les classes de caractères par les crochets. De même, le point d'interrogation signifie l'option, l'astérisque le fermeture de Kleene et le plus la fermeture positive.

Des commandes débutant par anti-oblique désigne certaines classes de caractères, qui peuvent aussi être prédéfinies par des noms que l'on peut utiliser avec la syntaxe `[:`$\langle nom \rangle$`:]`, comme dans `[:upper:]` pour la classe des caratères alphabétiques majuscules.

Commandes de Perl utilisant les expressions régulières

Parmi les commandes de Perl utilisant les expressions régulières, il y a le test d'appariement qui se présente de la manière suivante :

```
if ($reponse =~ m/^[0-9]+$/) {
    print "chiffres seulement\n"
} else {
    print "pas seulement des chiffres\n"
}
```

Le test d'égalité `=~` apparie le contenu de la variable `$reponse`, présumément une chaîne lue au terminal, avec l'expression régulière entre oblique ; le `m` devant la première oblique indiquant qu'il s'agit de faire une tentative d'appariement.

Les regroupements peuvent être utilisés en Perl, avec comme avantage que les portions de chaînes reconnues par chacun des groupes seront accessibles dans des variables nommées `$1`, `$2`, ... Ainsi, on peut écrire :

```
print "Entrez un montant et une monnaie : " ;
```

```
$reponse = <STDIN> ;      # lire la réponse au terminal
chomp($reponse) ;         # enlève la fin de ligne finale de $reponse
if ($reponse =~ m/([0-9]+\.[0-9]{2})([FE])/) {
    $montant = $1 ;
    $monnaie = $2 ;
    if ($monnaie eq "F") {    # conversion de francs en euros
        $nouvMontant = $montant / 6.55957 ;
        printf "%.2f euros\n", $nouvMontant ;
    } else {                  # conversion d'euros en francs
        $nouvMontant = $montant * 6.55957 ;
        printf "%.2f francs\n", $nouvMontant ;
    }
} else {
    print "Désolé, monnaie inconnue\n" ;
}
```

Notons tout particulièrement dans ce petit programme l'utilisation des variables `$1` et `$2` juste après l'appariement pour récupérer la portion de chaîne appariée par les deux regroupements dans l'expression régulière. Pour exécuter ce programme, il suffit de le transcrire dans une fichier, par exemple `conversion.prl`, puis d'utiliser la commande `perl` (sous Linux) :

```
> perl conversion.prl
Entrez un montant et une monnaie : 7.55F
1.15 euros
```

Il est aussi possible de remplacer le `m` dans la «comparaison» avec `=~` par un `s`, ce qui veut alors dire non plus de faire un simple appariement, mais de faire une substitution. Dans ce cas, on peut aussi utiliser les groupes. Supposons que vous ayiez saisi un texte en html où vous avez malencontreusement utilisé explicitement la mise en italique (<i>) plutôt que l'emphase (<em>). L'application de la substitution suivante remplace pour un texte les balises italiques par des balises d'emphase (en supposant que le texte en italique ne contient pas lui-même de balise) :

```
$texte =~ s/<i>([^<]*)<\/i>/<em>\1<\/em>/g ;
```

Les anti-obliques (\) servent ici à protéger les obliques (/) des balises pour que Perl ne les confonde pas avec les obliques délimitant l'expression régulière et le texte de remplacement.

Nouvelles extensions aux expressions régulières

Perl ajoute encore de nouvelles extensions aux expressions régulières qui ne font pas partie des expressions régulières au sens théorique du terme. L'une de ces extensions très puissantes est l'anticipation, c'est-à-dire la faculté de vérifier si un texte s'apparie à une expression régulière avant de procéder à un appariement partiel. La syntaxe est :

`(?=`*e*`)`

qui est interprétée comme le fait de vérifier si à une certaine position dans le texte il est possible d'apparier e, mais sans consommer effectivement de texte. Supposons par exemple que l'on veuille remplacer le point décimal dans les nombres d'un texte mais en évitant de

confondre le point final d'une phrase. Supposons que nous pouvons distinguer les deux cas par le fait qu'un point décimal est toujours suivi d'au moins un chiffre de 0 à 9. La commande suivante va faire la substitution :

```
$texte =~ s/([0-9]+)(?=\.[0-9]+)\.([0-9]+)/\1,\2/g ;
```

Après avoir apparié une séquence d'au moins un chiffre, on vérifie si on peut apparier un point suivi d'au moins un chiffre. Si oui, l'appariement se fait puis on utilise les deux portions de texte appariées pour recomposer le nombre avec la virgule décimale.

3.4 Le modèle objet de Python et Java

Java et Python partage une vision des expressions régulières modélisées par des objets. La proximité entre les deux nous incite à les traiter en parallèle.

Le langage Python est, comme Perl, l'un des langages de script les plus populaires à ce jour. L'avantage accordé à Python sur Perl vient du fait que Python est considéré comme un langage à objets, avec toutes les vertues associées à ce type de langages du point de vue structuration et réutilisation. Cela fait de Python, en plus d'un langage de script, un langage de prototypage apprécié, en particulier par les chercheurs dans les domaines du système, des langages de programmation ou dans le traitement des documents. Pour plus d'information sur Python, il est possible de consulter le site internet qui lui est consacré (voir [Pyt]). Ce site contient énormément d'informations sur le langages, et en particulier sa documentation et ses distributions.

Le langage Java n'a plus guère besoin d'être présenté, à ceci près que le paquetage des expressions régulières `java.util.regex` est livré en standard depuis la version 1.4.0 du langage. D'autres paquetages de traitement des expressions régulières existent et sont disponibles au téléchargement sur l'internet.

En Java, le paquetage `java.util.regex` propose deux classes pour gérer la plupart des fonctions sur les expressions régulières : `Pattern` et `Matcher`. La traitement par une expression régulière suppose généralement la séquence d'actions suivante :

1. L'inspection et la compilation de l'expression régulière en une forme interne qui va faire des appariements en distinguant majuscules et minuscules, ce qui donne un objet «patron» instance de la classe `Pattern`.
2. L'association d'un texte à apparier, une chaîne de caractères par exemple, ce qui donne un objet appariement instance de la classe `Matcher`.
3. L'exécution de l'appariement sur l'instance de `Matcher` pour vérifier si appariement il y a et en construire le résultat.
4. Si l'appariement s'est fait, rendre disponibles les informations sur cet appariement, dont le texte apparié par les regroupements.

Ces opérations sont réalisées par le protocole suivant dans le paquetage `java.util.regex` :

1. La méthode de classe `compile` de la classe `Pattern` prend en entrée une chaîne de caractères représentant l'expression régulière et retourne une instance de `Pattern` qui représente la forme interne pour l'expression régulière.

2. La méthode `matcher` de la classe `Pattern` prend une chaîne de caractères en argument et retourne une instance de la classe `Marcher` qui associe l'expression régulière au texte à apparier.
3. La méthode `find` de la classe `Matcher` tente un appariement du texte par l'expression régulière et retourne vrai un tel appariement existe et faux sinon.
4. Si l'appariement du texte par l'expression régulière a réussi, alors les méthodes `group`, `start` et `end` sans arguments (entre autres) de la classe `Matcher` permettent de récupérer respectivement le texte reconnu par l'expression régulière ainsi que les indices du début et de la fin de l'appariement dans le texte.

Il est possible de relancer l'appariement en envoyant à nouveau le message `find` à l'objet apparieur. Alternativement, la méthode `matches` vérifie si le texte entier est apparié par l'expression régulière (comme l'expression régulière avait été écrite pour apparier la position de début de chaîne au début et de fin de chaîne à la fin).

La classe `Matcher` défini également la méthode `groupCount` pour récupérer le nombre de regroupements qui ont été appariés au sein de l'expression régulière ainsi que la méthode `group(`i`)` pour récupérer le $i^{\text{ème}}$ groupe apparié. L'indice 0 désigne le groupe correspondant à toute l'expression régulière (comme si celle-ci était toujours entourée de parenthèses). Les méthodes `start(`i`)` et `end(`i`)` permettent de récupérer respectivement les indices de début et fin de l'appariement du $i^{\text{ème}}$ regroupement dans l'expression régulière.

La figure 3.2 présente la version Java de l'exemple de conversion de montants d'euros à francs et inversement déjà proposé en Perl. On note les deux étapes de compilation de l'expression régulière pour créer une instance de la classe `Pattern`, puis l'association à la chaîne à apparier en créant une instance de `Matcher` par l'envoi du message `matcher` à l'objet `Pattern`. La méthode `find` est utilisée pour procéder à l'appariement. Une fois l'appariement réalisé, s'il réussit, on utilise la méthode `group(int)` pour récupérer le texte apparié dans chacun des groupes. Java n'étant pas spécialement dédié au traitement des chaînes de caractères, c'est sans surprise que celui-ci est une peu plus complexe qu'en Perl, ce qui est visible dans l'apparition de conversions explicites de chaîne à réel.

L'utilisation de cette classe Java, après compilation demande de passer la chaîne du montant à traiter sur la ligne de commande :

```
> java -ea Conversion 10.65F
1,62 euros.
```

La version Python du même programme apparaît à la figure 3.3. Outre les aspects purement syntaxique de Python (l'indentation faisant partie intégrante de la syntaxe et la forme particulière de la structure de contrôle d'alternative), on reconnaît bien dans cet exemple la légèreté syntaxique des langages de scripts à la Perl mais avec un modèle objet du traitement des expressions régulières similaire à celui de Java. L'exécution d'un programme Python se fait en appelant la commande `python` avec le programme à exécuter en paramètre :

```
> python conversion.py
Entrez un montant et une monnaie : 10.65F
1.62 euros
```

```
import java.util.regex.Pattern ;
import java.util.regex.Matcher ;
import java.text.DecimalFormat ;

public class                    Conversion
{
  public static void          main(String[]   args) {
    String          expreg = "([0-9]+\\.[0-9]{2})([FE])" ;
    float           montant ;
    String          monnaie ;
    float           nouvMontant ;
    DecimalFormat   deuxChiffres = new DecimalFormat("#0.00") ;

    Pattern p = Pattern.compile(expreg) ;
    Matcher m = p.matcher(args[0]) ;

    try {
        if (m.find()) {
            montant = (new Float(m.group(1))).floatValue() ;
            monnaie = m.group(2) ;
            if (monnaie.equals("F")) {
                nouvMontant = montant / 6.55957f ;
                System.out.println(deuxChiffres.format(nouvMontant) +
                                   " euros.") ;
            } else {
                nouvMontant = montant * 6.55957f ;
                System.out.println(deuxChiffres.format(nouvMontant) +
                                   " francs.") ;
            } // end of if ()else
        } else {
            System.out.println("Désolé, monnaie inconnue : " +
                               args[0]) ;
        } // end of if ()else
    } catch (NumberFormatException e) {
        System.out.println(m.group(1) +
                           " ne représente pas un nombre.") ;
    } // end of try-catch

 } // ------------------------------------------------- end of main ()

} // *********************************************** classe Conversion
```

Fig. 3.2 – Conversion de monnaie en Java

```
import re      # module des expressions régulières

expreg = re.compile('([0-9]+\.[0-9]{2})([FE])')

line = raw_input('Entrez un montant et une monnaie : ')
m = expreg.match(line)
if m:
    montant = m.group(1)
    monnaie = m.group(2)
    if monnaie == 'F':
        nouvMontant = float(montant) / 6.55957
        print '%(nouvMontant)0.2f euros' % vars()
    else:
        nouvMontant = float(montant) * 6.55957
        print '%(nouvMontant)0.2f francs' % vars()
else:
    print 'Désolé, monnaie inconnue\n'
```

FIG. 3.3 – Conversion de monnaie en Python

Chapitre 4

Langages non-contextuels, grammaires et automates à pile

La deuxième catégorie de langages formels dans la hiérarchie de Chomsky est constituée des langages dits non-contextuels. Sauf pour ceux qui sont spécifiquement réguliers (selon l'inclusion du théorème), ces langages ne sont pas définissables par des expressions régulières. Une alternative est cependant proposée pour définir ces langages en intention : les grammaires non-contextuelles. Opérationnellement, la reconnaissance des chaînes faisant partie d'un langage non-contextuels est réalisable par un automate à pile. Ce chapitre présente ces différents concepts.

4.1 Grammaires non-contextuelles

Parmi tous les langages construits sur un alphabet donné, seule une petite partie peut être définie à partir d'opérations ensemblistes et donc d'expressions régulières telles que nous les avons vu au chapitre 2. Pour ceux qui ne font pas partie de cette classe des langages réguliers, les mêmes problèmes fondamentaux se posent encore :

1. Comment définir ces langages en intension ?
2. Comment décider si une chaîne $\omega \in \Sigma^*$ appartient au langage ou non ?

La catégorie des langages dits non-contextuels (ou «hors-contextes») est formée de langages dont la structure est telle qu'on peut découper ses chaînes en parties dont la structure est la même peu importe le contexte dans lequel celles-ci apparaissent dans la chaîne, à partir du moment où elles sont admises dans ce contexte.

Les langages non-contextuels sont définis en intension à l'aide de grammaires non-contextuelles. Informellement, une grammaire non-contextuelle est un ensemble de règles de construction des chaînes ou phrases d'un langage. Plus formellement, on la définit de la manière suivante :

> **Grammaire non-contextuelle :** une grammaire non-contextuelle est un quadruplet $G = (V, T, P, S)$ où
>
> V **:** est un ensemble fini de variables ou non-terminaux,
>
> T **:** est un ensemble fini de terminaux, c'est-à-dire l'alphabet du langage,

P **:** est un ensemble fini de règles de production de la forme $A \rightarrow \alpha$ où A est un symbole non-terminal et α est une séquence de symboles terminaux et non-terminaux,

S **:** est un symbole non-terminal appelé *axiome* à partir duquel toutes les chaînes du langage sont engendrées par application des règles de production.

Exemple 4.1 Les expressions arithmétiques simples. Les règles de production suivantes définissent une grammaire pour les expressions arithmétiques simples :

$$
\begin{array}{ll}
E \rightarrow E\ A\ E & A \rightarrow \texttt{+} \\
E \rightarrow \texttt{(}\ E\ \texttt{)} & A \rightarrow \texttt{-} \\
E \rightarrow \texttt{-}\ E & A \rightarrow \texttt{*} \\
E \rightarrow \texttt{id} & A \rightarrow \texttt{/} \\
 & A \rightarrow \texttt{\^{}}
\end{array}
$$

L'axiome de cette grammaire est E inclus dans l'ensemble $V = \{E, A\}$ des non-terminaux. L'ensemble des terminaux est $T = \{\texttt{(}, \texttt{)}, \texttt{id}, \texttt{-}, \texttt{+}, \texttt{*}, \texttt{/}, \texttt{\^{}}\}$. Lorsque plusieurs règles de production s'appliquent au même symbole non-terminal, on utilise généralement une forme abrégée avec la verticale d'alternative :

$$
\begin{array}{lcl}
E & \rightarrow & E\ A\ E \mid \texttt{(}\ E\ \texttt{)} \mid \texttt{-}\ E \mid \texttt{id} \\
A & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\^{}}
\end{array}
$$

□

L'application des règles de production permet de générer les phrases du langage. C'est la notion de dérivation :

Dérivation : (notée $\Rightarrow$) application d'un règle de production pour remplacer un non-terminal par l'une des parties droites de ses règles de production.

Exemple 4.2 Dans 'E', on peut remplacer le non-terminal E en utilisant la troisième règle de production de la grammaire précédente pour obtenir $\texttt{-}\ E$, ce que l'on note $E \Rightarrow \texttt{-}\ E$. On peut appliquer à nouveau une dérivation et ainsi de suite comme dans la séquence de dérivations suivante :

$$
E \Rightarrow \texttt{-}\ E \Rightarrow \texttt{-}\ \texttt{(}\ E\ \texttt{)} \Rightarrow \texttt{-}\ \texttt{(}\ E\ A\ E\ \texttt{)} \Rightarrow \texttt{-}\ \texttt{(}\ \texttt{id}\ A\ E\ \texttt{)} \Rightarrow \texttt{-}\ \texttt{(}\ \texttt{id} + E\ \texttt{)} \Rightarrow \texttt{-}\ \texttt{(}\ \texttt{id} + \texttt{id}\ \texttt{)}
$$

□

En termes de vocabulaire et de notations, on dira que E se dérive en $\texttt{-}\ E$ noté $E \Rightarrow \texttt{-}\ E$. L'application de 0, 1, ou plusieurs dérivations se note $\overset{*}{\Rightarrow}$. On pourra donc écrire que $E \overset{*}{\Rightarrow} \texttt{-}\ \texttt{(}\ \texttt{id} + \texttt{id}\ \texttt{)}$. Lorsqu'au moins une dérivation est faite, on utilisera la notation $\overset{+}{\Rightarrow}$ qui rappelle la fermeture positive par rapport à la fermeture de Kleene dans les expressions régulières. Ainsi, on a $E \overset{+}{\Rightarrow} \texttt{-}\ \texttt{(}\ \texttt{id} + \texttt{id}\ \texttt{)}$. On introduit également les notions de *protophrase* et *phrase* :

Protophrase : séquence de symboles terminaux et non-terminaux obtenue par dérivation à partir de l'axiome.

Phrase : séquence de symboles terminaux obtenue par dérivation à partir de l'axiome.

Les notions de dérivation, de protophrase et de phrase nous amènent à définir la notion de langage défini par une grammaire :

Langage accepté par une grammaire G **:** Soit $G = (V, T, P, S)$ une grammaire non-contextuelle, le langage défini par cette grammaire, noté $L(G)$, est l'ensemble des phrases $\omega \in T^*$ telles que $S \overset{+}{\Rightarrow} \omega$.

Théorème 4.1 *Un langage $L(G)$ défini par une grammaire non-contextuelle G est un langage non-contextuel.*

Comme nous l'avons rappelé en début de chapitre, il se pose pour les langages non-contextuels les mêmes questions fondamentales que pour tous les langages formels. Nous venons de voir comment définir en intension un langage non-contextuel en utilisant une grammaire non-contextuelle. Comment vérifier qu'une phrase appartient à $L(G)$? Pour cela, il faut trouver une séquence de dérivations de la phrase depuis l'axiome de G. Pendant ce processus, à chaque application d'une dérivation implique deux choix s'imposent :

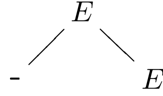
1. Quel non-terminal réduire ?
2. Quelle règle de production utiliser pour réduire le non-terminal choisi ?

En fait, dans une protophrase, il y a souvent plusieurs non-terminaux candidats à la réduction. Tous peuvent également être choisis pour effectuer la dérivation. Il se trouve qu'il a été démontré que ce choix n'influence pas le fait de trouver ou non une séquence de dérivation permettant d'engendrer une phrase. Le choix du non-terminal se fait donc selon une stratégie à déterminer. Les deux stratégies les plus utilisées sont :
- la dérivation à gauche, notée $\underset{g}{\Rightarrow}$ où on choisit toujours le non-terminal le plus à gauche dans la protophrase, et
- la dérivation à droite, notée $\underset{d}{\Rightarrow}$ où on choisit toujours le non-terminal le plus à droite dans la protophrase.

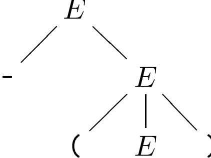
Pour comprendre et manipuler les dérivations, il est souvent utile de passer par une représentation graphique, plus précisément un arbre de dérivation :

Arbre de dérivation : L'arbre de dérivation d'une phrase est obtenu d'une dérivation $A \Rightarrow \alpha_1 \Rightarrow \alpha_2 \ldots \Rightarrow \alpha_n$ de la manière suivante :
- La racine de l'arbre de dérivation est A.
- Soit une protophrase $\alpha_i = X_1 \ldots X_j \ldots X_k$ et $X_j \rightarrow X_{j_1} \ldots X_{j_m}$ la règle de production appliquée pour obtenir α_{i+1}, alors l'arbre de dérivation à l'étape $i+1$ est obtenu de l'arbre à l'étape i en ajoutant m fils $X_{j_1} \ldots X_{j_m}$ à la feuille X_j.

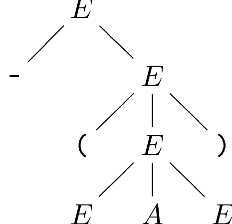
Exemple 4.3 Construisons l'arbre pour la séquence de dérivations précédente de la phrase `-(id+id)`. Selon la première règle, la racine de l'arbre est E. L'application de la première dérivation $E \Rightarrow$ - E nous donne l'arbre :



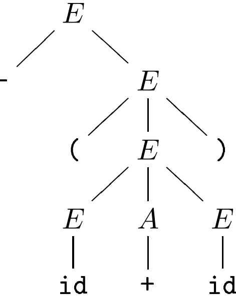
La troisième étape de dérivation `-` $E \Rightarrow$ `- (` E `)` construit les fils de la feuille E :



À la quatrième étape, `- (` E `)` $\Rightarrow$ `- (` $E\ A\ E$ `)`, on passe à :



Les trois étapes suivantes, `- (` $E\ A\ E$ `)` $\Rightarrow$ `- ( id` $A\ E$ `)` $\Rightarrow$ `- ( id +` E `)` $\Rightarrow$ `- ( id + id )`, font apparaître les trois feuilles de symboles terminaux :



□

À la question de savoir «Comment décider si une phrase $\omega \in L(G)$?», on répond qu'il faut trouver une dérivation à partir de l'axiome de la grammaire G ou alors construire un arbre de dérivation, aussi appelé arbre d'analyse, pour cette phrase. Parmi les applications les plus importantes des grammaires non-contextuelles se trouve la définition de la syntaxe des langages de programmation. La partie du compilateur qui se charge de vérifier si un programme est conforme à la syntaxe du langage fait exactement cela : il construit un arbre d'analyse pour le programme, arbre qui sert non seulement à montrer que le programme est syntaxiquement correct, mais sert aussi de base aux traitements subséquents réalisés par le compilateur.

4.2 Automate à pile

L'automate à pile est aux grammaires non-contextuelles ce que les automates d'états finis sont aux expressions régulières. Informellement, un automate à pile est un automate d'états fini auquel est adjoint une pile. Plus formellement, il est défini de la manière suivante :

Automate à pile : un automate à pile est une structure $AP = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ où :

Q est un ensemble fini d'états.

Σ est l'alphabet fini d'entrée.

Γ est l'alphabet fini de pile.

δ est une fonction de transition depuis $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ vers les sous-ensembles finis de $Q \times \Gamma^*$.

$q_0 \in Q$ est l'état initial.

$Z_0 \in \Gamma$ est le symbole placé initialement dans la pile.

$F \subseteq Q$ est le sous-ensemble des états finaux.

La fonction de transition est utilisée de la manière suivante. Un symbole est dépilé et un symbole de l'entrée est consommé ou un ϵ-transition est déclenchée et l'automate effectue une transition vers un des états spécifiés par la fonction de transition, puis 0, 1 ou plusieurs symboles également spécifiés par la fonction de transition sont empilés.

> **Comportement d'acceptation :** le comportement d'acceptation d'un automate à pile AP consiste à consommer toute la chaîne d'entrée ω et à parvenir dans un état final $f \in F$ alors que la pile de l'automate est vide ; la chaîne ω est alors dite acceptée par l'automate AP.

Comme pour les automates d'états finis, le comportement d'acceptation de l'automate à pile nous amène à définir la notion de langage accepté par un automate à pile :

> **Langage accepté par un automate à pile :** Soit $AP = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un automate à pile, alors le langage accepté par AP, noté $L(AP)$ est l'ensemble des chaînes $\omega \in \Sigma^*$ telle que ω est acceptée par AP.

Ce qui nous permet d'énoncer le théorème d'équivalence entre les automates à pile et les grammaires non-contextuelles :

Théorème 4.2 *(équivalence grammaire non-contextuelle/automate à pile) Soit G une grammaire non-contextuelle définissant le langage $L(G)$, alors il existe un automate à pile AP tel que $L(AP) = L(G)$ et réciproquement.*

Le problème qui va maintenant se poser, comme il s'est posé pour les automates d'états finis par rapport aux expressions régulières, est de savoir comment construire l'automate à pile reconnaissant le langage défini par une grammaire non-contextuelle G. Un automate à pile analyseur pour une grammaire non-contextuelle définit une procédure par laquelle une dérivation ou un arbre d'analyse seront construits pour une phrase de manière à vérifier si celle-ci appartient au langage défini par la grammaire. Plusieurs difficultés se posent. D'abord, le choix de la règle de production à appliquer à chaque étape de dérivation peut induire des échecs dans la construction de la dérivation. Une stratégie de construction de la dérivation ou de l'arbre doit aussi être choisie : partir de l'axiome pour aller vers la phrase ou l'inverse ? Différentes formes de grammaires sont mieux adaptées à l'une ou à l'autre de ces stratégies. Enfin, la performance des analyseurs est un point crucial pour les applications.

Il existe en fait plusieurs méthodes pour construire des analyseurs. Certaines sont universelles, au sens où elles s'appliquent à toutes les grammaires non-contextuelles. Le défaut de ces méthodes est qu'elles ne produisent pas des analyseurs efficaces (temps d'analyse dans $0(n^3)$ où n est la taille de la phrase à analyser). D'autres méthodes produisent des analyseurs efficaces, mais elles ne sont pas universelles. Pourtant les classes de grammaires traitées par

ces méthodes sont souvent relativement riches, et suffisantes dans les applications pratiques (comme par exemple définir la syntaxe d'un langage de programmation). Ces méthodes sont présentées au chapitre 6.

4.3 Transformations de grammaires

Dans cette section sont présentés deux algorithmes relativement simples pour transformer des grammaires sans changer le langage reconnu. Ces transformations peuvent rendre des grammaires analysables par des méthodes efficaces.

4.3.1 Factorisation à gauche

Lorsqu'un analyseur construit une dérivation, il doit choisir à chaque étape le non-terminal à remplacer et la règle de production à utiliser pour le remplacer. Le choix de la règle de production pouvant causer des problèmes (toute séquence de choix ne produisant pas nécessairement une dérivation de la chaîne, même si une dérivation existe bel et bien), il vaut mieux avoir une grammaire où il existe un minimum d'ambiguité dans le choix des règles de production lorsqu'on connait les prochains symboles terminaux à dériver à partir du non-terminal à remplacer.

Un cas qui peut se produire est celui de multiples règles de production partageant un préfixe commun. Considérons par exemple une hypothétique grammaire des énoncés d'alternatives dans un langage de programmation :

$$\begin{array}{rcl} S & \rightarrow & \texttt{if}\ E\ \texttt{then}\ S \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \ldots \\ E & \rightarrow & \ldots \end{array}$$

On constate ici que les deux règles de production pour l'alternative partagent plusieurs symboles en préfixe, et en plus qu'elles commencent toutes les deux par le même terminal (`if`). Pour éviter cette situation, il suffit de factoriser le préfixe commun en créant un nouveau non-terminal retardant le choix entre les deux terminaisons au moment où le préfixe commun aura été reconnu.

L'algorithme de factorisation à gauche est donné à la figure 4.1. Il consiste à traiter tour à tour tous les non-terminaux, et pour chacun de ces derniers, à identifier les préfixes communs, à les extraire des productions concernées, à introduire un nouveau non-terminal dans chaque cas à la fin du préfixe commun ainsi extrait et à définir pour le nouveau non-terrminal autant de règles de production avec chaque suffixe distinct.

Exemple 4.4 Pour la grammaire des alternatives précédente, on peut identifier comme préfixe commun la portion $\alpha = \texttt{if}\ E\ \texttt{then}\ S$. Ceci nous laisse les suffixes $\beta_1 = \epsilon$ et $\beta_2 = \texttt{else}\ S$. Les productions non-concernées par la factorisation ne sont pas connues ici, mais disons que $\gamma = \ldots$. On obtient alors la grammaire transformée :

$$\begin{array}{rcl} S & \rightarrow & \texttt{if}\ E\ \texttt{then}\ S\ S' \mid \ldots \\ S' & \rightarrow & \epsilon \mid \texttt{else}\ S \\ E & \rightarrow & \ldots \end{array}$$

□

Soit une grammaire G, l'algorithme produit une grammaire G' équivalente où tous les préfixes communs les plus longs entre les parties droites de productions auront été factorisés.

1. **tant que** il existe des productions avec préfixe commun **faire**
2. **Pour** chaque non-terminal A **faire**
3. trouver le plus long préfixe α commun à deux alternatives de productions ou plus.
4. **si** $\alpha \neq \epsilon$ **alors**
5. remplacer toutes les A-productions $A \rightarrow \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_n|\gamma$, où γ représente toutes les alternatives qui ne commencent pas par α, par les productions :
   $A \rightarrow \alpha A'|\gamma$
   $A' \rightarrow \beta_1|\beta_2|...|\beta_n$

fin

fin

FIG. 4.1 – Algorithme de factorisation à gauche

4.3.2 Elimination des récursivités à gauche

Par grammaire récursive à gauche, on entend :

> **Grammaire récursive à gauche :** une grammaire est dite récursive à gauche si elle admet au moins une séquence de dérivations de la forme $A \stackrel{*}{\Rightarrow} A\alpha$ où A est un non-terminal.

En fait, le concept de récursivité à gauche est tout à fait similaire à la récursivité telle qu'on l'apprend dans les langages de programmation : un non-terminal se dérive un 1 ou plusieurs étapes en une séquence de symboles terminaux et non-terminaux où il apparaît en première position. Si une telle dérivation existe, il est possible de répéter la séquence d'applications de règles de production à nouveau sur le A en première position de la protophrase résultante, et ainsi de suite *ad infinitum*.

Certaines des approches d'analyse que nous verrons au chapitre 6 ne s'appliquent pas au cas des grammaires récursives à gauche. La possibilité de transformer systématiquement une grammaire G récursive à gauche en une grammaire G' équivalente qui ne soit pas récursive à gauche est donc intéressant dans ces cas de figure.

Remarquons qu'il ne s'agit pas uniquement de récursivité à gauche immédiate, c'est-à-dire où le symbole non-terminal apparaît au début de l'une des parties droites de ses propres productions. Non, il s'agit plus généralement d'éliminer toute récursivité à gauche dans la grammaire, qu'elle soit directe ou indirecte.

Exemple 4.5 La grammaire suivante est récursive à gauche :

$$E \rightarrow E + E$$

puisqu'elle contient une récursivité à gauche immédiate sur le non-terminal E. La grammaire suivante est également récursive à gauche mais ne possède pas de récursivités à gauche immé-

diates :

$$\begin{array}{lcl} S & \rightarrow & A\alpha \\ A & \rightarrow & S\beta \mid \ldots \end{array}$$

puisqu'on peut dériver $S \stackrel{+}{\Rightarrow} S\beta\alpha$. □

Considérons d'abord une méthode d'élimination des récursivités à gauche immédiates. La récursivité à gauche immédiate se présente dans une grammaire sous la forme suivante :

$$A \rightarrow A\alpha \mid \beta$$

où β est une séquence de symboles ne débutant pas par A. Pour comprendre comment éliminer la résursivité à gauche immédiate, il faut examiner les dérivations possibles à l'aide de ses règles de production. Partant de A, on peut obtenir :

$$\begin{array}{lllllllll} A & \Rightarrow & \beta \\ A & \Rightarrow & A\alpha & \Rightarrow & \beta\alpha \\ A & \Rightarrow & A\alpha & \Rightarrow & A\alpha\alpha & \Rightarrow & \beta\alpha\alpha \\ A & \Rightarrow & A\alpha & \Rightarrow & A\alpha\alpha & \Rightarrow & A\alpha\alpha\alpha & \Rightarrow & \ldots \end{array}$$

On se rend donc compte que les séquences qui peuvent être dérivées à partir de A sont formées d'une apparition de β suivie de 0, 1 ou plusieurs répétitions de α. Construisons donc une grammaire produisant le même langage et qui ne soit pas récursive à gauche mais plutôt récursive à droite :

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$

La ligne (5) dans l'algorithme de la figure 4.2 généralise cette idée pour toute récursivité à gauche immédiate, c'est-à-dire lorsque les productions sont de la forme :

$$A_i \rightarrow A_i\alpha_1 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

que l'on remplace par :

$$\begin{array}{lcl} A_i & \rightarrow & \beta_1 A'_i \mid \ldots \mid \beta_n A'_i \\ A'_i & \rightarrow & \alpha_1 A'_i \mid \ldots \mid \alpha_m A'_i \mid \epsilon \end{array}$$

Exemple 4.6 Considérons la grammaire des expressions arithmétiques suivante :

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \texttt{id} \end{array}$$

Pour éliminer la récursivité à gauche immédiate sur E, on prend $\beta = T$ et $\alpha = + T$ pour obtenir :

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \end{array}$$

De même, pour éliminer la récursivité à gauche immédiate sur le symbole T, on prend $\beta = F$

Soit une grammaire G sans cycle (dérivation de la forme $A \stackrel{*}{\Rightarrow} A$) et sans production vide (i.e. de la forme $A \rightarrow \epsilon$), l'algorithme produit une grammaire G' équivalente sans récursivité à gauche.

1. On ordonne les non-terminaux A_1, A_2, ..., A_n.
2. **pour** $i := 1$ jusqu'à n **faire**
3. **pour** $j := 1$ jusqu'à $i - 1$ **faire**
4. remplacer toutes les productions $A_i \rightarrow A_j\gamma$ par les productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|...|\delta_k\gamma$, où $A_j \rightarrow \delta_1|\delta_2|...|\delta_k$ sont toutes les A_j-productions courantes.

   fin
5. éliminer les récursivités à gauche immédiates des A_i-productions à l'aide de la transformation suivante :

   $A_i \rightarrow A_i\alpha_1|...|A_i\alpha_m|\beta_1|...|\beta_n$

   devient :

   $A_i \rightarrow \beta_1 A_i'|...|\beta_n A_i'$

   $A_i' \rightarrow \alpha_1 A_i'|...|\alpha_m A_i'|\epsilon$

fin

FIG. 4.2 – Elimination des récursivités à gauche

et $\alpha = * F$ pour obtenir :

$$
\begin{array}{rcl}
T & \rightarrow & FT' \\
T' & \rightarrow & * FT' \mid \epsilon
\end{array}
$$

□

L'application de l'algorithme de la figure 4.2 vise à éliminer les récursivités à gauche autant directes qu'indirectes. Le principe de l'algorithme consiste à ordonner les non-terminaux de 0 à n puis à les traiter dans l'ordre croissant de manière à ce qu'à l'étape i, toutes les récursivités à gauche immédiates de même que toutes les récursivités à gauche indirectes qui, débutant par un non-terminal d'indice $j \leq i$, passent par un non-terminal $k \leq i$. Cet invariant étant maintenu à chaque itération, lorsque i devient égal à n, toutes les récursivités à gauche directes ou indirectes auront été éliminées.

Pour arriver à cet objectif, l'idée à chaque étape est d'utiliser le fait que si on a une grammaire où $A \rightarrow B\alpha$ et $B \rightarrow \beta$, alors la grammaire obtenue en remplaçant le B au début de la partie droite de la production sur A par la partie droite de la production sur B, c'est-à-dire avec la production $A \rightarrow \beta\alpha$ est équivalente à la grammaire initiale. Par ce genre de remplacement, on fait en sorte que toute récursivité indirecte à l'étape i passant par un non-terminal d'indice $j \leq i$ devient en fait une récursivité directe. Cette récursivité directe est alors éliminée simplement par la transformation générale précédente. L'algorithme de la figure 4.2 formalise cette idée.

Exemple 4.7 Considérons la grammaire suivante :

$$
\begin{array}{rcl}
S & \rightarrow & A\mathtt{a} \mid \mathtt{b} \\
A & \rightarrow & A\mathtt{c} \mid S\mathtt{d}
\end{array}
$$

Ordonnons les non-terminaux en considérant que S a l'indice 1 et A a l'indice 2.

Itération 1 — $i = 1$

Comme il n'y a pas de non-terminaux précédents, il suffit d'éliminer les récursivités à gauche immédiate sur le symbole S. Comme il n'y a pas de telle récursivité à gauche immédiate, on ne change pas la grammaire.

Itération 2 — $i = 2$

La production $A \rightarrow S\texttt{d}$ fait apparaître un non-terminal déjà traité à gauche de la partie droite de la règle. On le remplace donc par toutes les parties droites des productions courantes sur ce symbole, ce qui nous donne les productions suivantes pour A :

$$A \rightarrow A\texttt{c} \mid A\texttt{ad} \mid \texttt{bd}$$

Ensuite, éliminons les récursivités immédiates sur A. Pour cela, on a $\alpha_1 = \texttt{c}$, $\alpha_2 = \texttt{ad}$ et $\beta_1 = \texttt{bd}$. On obtient donc les règles de production suivantes qui remplace les règles de production sur A précédentes dans la grammaire résultante :

$$\begin{array}{lcl} A & \rightarrow & \texttt{bd}A' \\ A' & \rightarrow & \texttt{c}A' \mid \texttt{ad}A' \mid \epsilon \end{array}$$

□

Chapitre 5

Le langage XML

Dans ce chapitre, nous allons introduire les concepts du langage XML, dont le balisage structurel ainsi que les descriptions de documents par DTD et par schémas. Nous allons également faire un parallèle entre les modes de descriptions de documents XML et les grammaires et entre les documents XML et les arbres d'analyse pour des phrases de langage non-contextuels.

5.1 Langage à balises sémantiques

XML est un langage standard de description et d'échanges de documents structurés. Il est issu d'une longue tradition fondée à la fois dans les standards de descriptions de données complexes ayant donné naissance au langage SGML et dans les standards de description de la présentation des documents dont est issu HTML. XML est standardisé par le W3C.

XML est fondé sur un concept très simple : le *balisage structurel*. L'idée du balisage structurel est de rendre explicite la structure d'un document en découpant ce document en parties marquées par des balises qui indiquent le début et la fin de chacune de ces parties et sous-parties. Dans la terminologie XML, une partie de document balisée s'appelle *élément*. Une balise est notée entre symboles plus petit '<' et plus grand '>' et porte un nom. Une balise ouvrante ne porte que le nom de l'élément alors qu'un balise fermante porte le nom de l'élément précédé d'une oblique '/'. Par exemple, des éléments dénommés `nom` et `adresse` seront «parenthésés» par les balises suivantes :

`<nom> ... </nom>`

`<adresse> ... </adresse>`

Cette notion de balisage n'est pas propre à XML, puisqu'elle était déjà présente en SGML et en HTML, qui est largement plus connu. De fait, XML fait un usage du balisage différent de celui d'HTML. L'objectif d'HTML est d'indiquer à un programme comment présenter un document à l'écran ou le produire sur papier. Le balisage d'HTML sert donc à indiquer des modes et des effets de présententation, comme la mise en italique ou en gras, ou encore la mise en paragraphe, etc.

En XML, le balisage est utilisé pour segmenter un document en informations sémantiquement cohérentes. C'est ainsi qu'en XML, on parle de *balisage sémantique*. Grâce au balisage sémantique, XML impose une indépendance entre contenu et présentation. Le balisage sémantique permet d'identifier les types d'information contenues dans un fichier, types qui peuvent ensuite servir à décider comment les présenter, que ce soit en les transformant en HTML pour

un affichage à l'écran, en postscript pour l'impression sur papier, ou en son numérique pour reproduction par synthèse de son via des enceintes.

Exemple 5.1 En balisage sémantique, on pourra marquer une date par des balises de nom `date` en choisissant un format standard comme :

```
<date>1961-05-31</date>
```

Selon l'utilisation de cette information, dans un navigateur internet ou sur papier ou encore sur téléphone portable avec synthèse vocale, cette information pourra être présentée par 31/05/1961, ou encore 31 mai 1961, on encore par une voix disant cette date. □

Parmi les présentations possibles d'un document XML, citons la présentation sous forme de page web, sous forme de document papier, sous forme d'image, sous forme de synthèse vocale, etc. L'idée d'XML de séparer identification du contenu de sa présentation suppose que l'information sur la présentation, qui n'est pas dans le fichier lui-même, sera apportée de manière indépendante du document par une transformation d'un document source XML en une forme présentable (document HTML, document postscript, fichier de son, etc.). Nous verrons l'aspect transformation de document au chapitre 8.

Documents XML

Un document XML est un fichier dont le contenu est balisé selon les règles du langage XML. Le balisage peut être introduit très libéralement en XML, en définissant chaque fois que nécessaire des noms d'éléments appropriés à l'identification de leur contenu. Les règles syntaxiques d'XML imposent des contraintes sur la formation des noms d'éléments (séquence de caractères alphanumériques, des caractères tiret et souligné ou du point commençant par un caractère alphabétique ou un souligné ; le deux-points peut aussi être utilisé mais il a une signification spécifique) ainsi que la stricte imbrication des balises (parenthésage correctement imbriqué des balises ouvrantes et fermantes), incluant l'obligation de fermer explicitement toute balise ouvrante (sauf le cas particulier des éléments sans contenu pour lequels une forme abrégée existe, comme nous le verrons plus loin).

> **Document bien formé :** un document XML est dit bien formé s'il respecte toutes les règles syntaxiques d'XML.

De plus, XML permet de définir des types de documents. Un type de document définit un certain nombre de noms d'éléments ainsi que le contenu de ces éléments (autres éléments, données élémentaires, ou séquence de caractères non analysée). Ces types de documents sont définis par ce que l'on appelle des DTD («*Document Type Definition*») ou encore selon la norme plus récente par des schémas XML (XML-Schema). Nous reviendrons sur ces formes de définition dans les sections suivantes.

> **Document valide :** un document XML est dit valide s'il est bien formé et s'il respecte la structure définie par sa DTD ou son shéma.

La structure d'un document XML bien formé s'établit comme suit. Tout document commence par une partie appelée *prologue* qui consiste en un ensemble de déclarations de base sur le document (par exemple, la version d'XML et le jeu de caractères utilisés). Ensuite, le document comporte nécessairement un et un seul arbre d'éléments. La structure arborescente est induite par l'imbrication stricte des éléments. Le fait qu'il n'y ait qu'un seul arbre dans

un document est une restriction d'XML. Ceci implique que le contenu en tant que tel d'un document est toujours balisé comme un et un seul élément (pouvant bien sûr en contenir plusieurs). Enfin, un document XML peut contenir des commentaires, dans la mesure où on respecte par ailleurs les règles syntaxiques d'XML. Un élément de commentaire a la forme suivante :

```
<!-- ...>
```

La déclaration XML débute le document. Elle indique au processeur de document la version d'XML utilisée, le jeu de caractères et le fait que le document soit interprétable en lui-même ou s'il dépend d'autres documents, comme une description de sa structure. La forme de la déclaration XML est :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
```

Cette balise est d'un type spécial (introduit par le point d'interrogation) et n'a pas besoin d'être fermée. Il s'agit d'une balise d'instruction de traitement, sur lesquelles nous reviendrons plus loin. Le nom de la balise est `xml`. Les noms comme `version` ou `encoding` sont des attributs associés à la balise. Les chaînes entre guillemets sont les valeurs de ces attributs. Toute valeur d'attribut doit être placée entre guillemets ou entre apostrophes.

Après la déclaration XML au début du document, le prologue peut contenir une déclaration de type de document. Cette déclaration donne un nom au type de document et déclare les éléments pouvant apparaître dans le document de même que leur imbrication respective. La forme de cette déclaration est :

`<!DOCTYPE rapport [` *déclarations* `]>`

Nous verrons en détail la partie «déclarations» lorsque nous décrirons les DTD à la prochaine section. Après cette déclaration de type de document, on trouve l'arbre d'éléments, donc le contenu comme tel du document placé entre une paire de balise ouvrante et fermante dont le nom d'éléments est le nom de type de document apparaissant dans la déclaration de type de document (dans notre exemple, ce serait `rapport`).

Les critères qui font qu'un document XML est bien formé sont :
- respect des règles de nommage des éléments (balises),
- respect du pairage entre balises ouvrantes et fermantes (modulo les balises spéciales ne nécessitant pas de fermeture),
- respect de l'imbrication stricte des balises, et
- respect de l'unicité de l'arbre d'éléments formant le contenu du document.

Les balises définies par le créateur du document (ou plus loin de la DTD ou du schéma) peuvent, comme les balises prédéfinies, porter des attributs. Les attributs sont placés dans la balise ouvrante d'un élément. Par exemple, la balise ouvrante `rapport` de notre document précédent pourrait se présenter de la manière suivante :

```
<rapport langue="FR" date-modif="08.05.2002" diffusion="confidentiel">
```

Les noms d'attributs respectent des règles de construction identiques à celles des noms d'éléments.

Exemple 5.2 Le document XML suivant ne comporte pas de déclaration de type de document ; il est donc bien formé sans pouvoir cependant être validé. Il contient des informations sur un parc d'ordinateurs d'une entreprise.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<ordinateurs>
```

```
  <machine>
    <nom>malin.univ-ubs.fr</nom>
    <no-ethernet>09:F6:AA:08:1B:65</no-ethernet>
    <no-ip>185.90.8.33</no-ip>
    <date-mise-en-service>1997-03-17</date-mise-en-service>
    <type-systeme>Windows 95</type-systeme>
    <installateur>Jean Le Breton</installateur>
  </machine>
  <machine>
    <nom>recent.univ-ubs.fr</nom>
    <no-ethernet>12:98:B6:1A:F0:08</no-ethernet>
    <no-ip>193.8.233.6</no-ip>
    <date-mise-en-service>2001-10-12</date-mise-en-service>
    <type-systeme>Linux Redhat 7.1</type-systeme>
    <installateur>Titouan Le Minutieux</installateur>
  </machine>
</ordinateurs>
```

□

5.2 Types de documents et DTD

Les documents XML valides obéissent à des règles de construction précises, c'est-à-dire à un type de documents. XML propose deux moyens pour définir des types de document. Le plus ancien et le plus simple est la DTD ou «*Document Type Definition*». Plus récemment, une nouvelle forme de définition est apparue avec les schémas, ou encore *XML-Schemas*, qui sont une extension des DTD permettant une définition plus précise. DTD et schémas doivent être définies en utilisant un certain langage. On aurait pu choisir n'importe quelle syntaxe pour ce faire, mais XML étant lui-même un langage extensible par définition de nouveaux types de documents, on a choisi de définir les DTD et les schémas comme des documents XML, possédant donc leurs propres types de documents, et donc leurs propres types d'éléments.

Une DTD définit les types d'éléments, donc les noms de balises, le type de leur contenu et l'organisation des types d'éléments les uns par rapport aux autres. Par organisation mutuelle des types d'éléments, on entend l'inclusion de types d'élements dans d'autres types d'éléments, y compris la multiplicité éventuelle des éléments pouvant apparaître dans d'autres éléments.

La DTD est introduite par la déclaration `DOCTYPE` dans le prologue du document. La déclaration `DOCTYPE` définit à la fois le nom du type de document auquel obéit le document courant, puis introduit les déclarations d'organisation propres à ce type de documents. Ces déclarations d'organisation se présentent sous une forme qui ressemble beaucoup à une grammaire non-contextuelle. En fait, le langage de définition de la grammaire du document est légèrement étendu pour inclure des constructions de grammaires EBNF («*Extended Backus-Naur Form*»). Pour chaque élément introduit par son nom, la déclaration donne le contenu de l'élément par son type de données prédéfinies ou par les noms d'éléments pouvant y apparaître. L'ordre et la multiplicité des élements dans cette déclaration est significative, comme dans une règle de production d'une grammaire. Par ailleurs, la déclaration d'un élément inclut également la définition des attributs que l'on pourra trouver dans la balise ouvrante de l'élément, attributs définis par leurs noms et les types de leur contenu.

Exemple 5.3 Considérons d'abord le prologue d'un document contenant un seul élément nommé `exemple` dont le contenu sera simplement du texte pur considéré comme non-structuré. Le prologue sera alors :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE exemple [
    <!ELEMENT exemple (#PCDATA)>
]>
<exemple>Ceci est un exemple de document XML</exemple>
```

Cet extrait forme un document XML complet, bien formé et valide. La balise `ELEMENT` fait partie du langage de définition des DTD. Elle introduit la définition d'un élément en donnant son nom (ici `exemple`) puis son contenu entre parenthèses (ici simplement le type de données de base `#PCDATA`, c'est-à-dire «*Parsed Character DATA*» dans le langage de définition des DTD). □

Une DTD peut être introduite dans la déclaration `DOCTYPE` *in situ* ou encore être définie dans un document à part qui est référencé dans la déclaration `DOCTYPE`. Supposons par exemple que la DTD du document exemple précédent soit définie dans un document `exemple.dtd`, alors la déclaration `DOCTYPE` suivante va faire référence à cette DTD externe :

```
<!DOCTYPE exemple SYSTEM 'exemple.dtd'>
```

Dans ce cas, on suppose que le fichier `exemple.dtd` est colocalisé avec le fichier `exemple.xml` contenant le document exemple précédent. XML définit des moyens pour référer à des DTD situées dans des bibliothèques de DTD, moyens que nous ne verrons pas ici (consultez [Mic01] pour plus de détails sur les utilisations pratiques de XML). De plus, lorsqu'un document XML utilise une DTD externe, il n'est plus interprétable de manière indépendante ; il faut donc modifier la déclaration `xml` pour refléter cela en changeant la valeur de l'attribut `standalone` :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
```

Contenu de la DTD

La DTD contient d'abord la déclaration des éléments du type de documents. Cette déclaration, comme nous l'avons vu, est faite par une balise spéciale `ELEMENT` dont la forme générale est :

<!ELEMENT *nom modèle* >

où *nom* est le nom de l'élément défini et *modèle* explicite le contenu de l'élément. Le contenu de l'élément est une liste d'éléments fils ou de types de données prédéfinis, ou encore l'un des mots-clés `ANY` ou `EMPTY` :

- Les éléments fils sont simplement introduits par leurs noms. Si les noms des éléments fils sont séparés par des virgules alors l'ordre dans lequel les éléments fils doivent apparaître est imposé, Par contre, si on les sépare par des barres verticales alors il s'agit d'une alternative, un élément parmi les alternatives devant apparaître.
- Les éléments feuilles ou les éléments mixtes (avec éléments fils et des données de base) peuvent contenir essentiellement un flôt de caractères dont le type prédéfini est `#PCDATA`. Dans un tel contenu peuvent apparaître tous caractères sauf les caractères '<' (qui indiquerait le début de la balise fermante) et le '&'. Des échappements sont prévus lorsqu'on doit utiliser ces caractères. D'autres types de contenus sont possibles, en utilisant la notion de *notation* que nous ne verrons pas ici.

- Le mot-clé ANY permet de définir un élément dont le contenu est libre (y compris avec des balises internes). Ce type d'éléments est particulièrement utile dans les phases initiales de définition d'une DTD pour reconnaître des documents existants mais dont le type n'est pas encore complètement défini.
- Le mot-clé EMPTY permet de définir des éléments dont le contenu sera toujours vide. Ce genre d'éléments porte normalement des attributs qui lui donnent un rôle particulier dans un document.

On peut utiliser dans la définition des éléments fils des opérateurs d'occurrence ou de multiplicité très similaire à ce que l'on rencontre dans les expressions régulières : '?' indique l'optionnalité, '*' indique la répétabilité optionnelle, alors que '+' indique la répétabilité avec présence d'au moins une occurence. Les parenthèses peuvent être utilisées pour délimiter la portée des opérateurs d'occurrence. Dans les définitions :

```
<!ELEMENT section (titre-section, corps-section)>
<!ELEMENT chapitre (titre, intro, section+)>
```

on indique qu'une section comporte un titre et un corps, alors qu'un chapitre est composé d'un titre, d'une introduction et d'au moins une section. Comme déjà mentionné, la barre verticale '|' introduit une alternative. Dans la définition :

```
<!ELEMENT corps-section (#PCDATA | bibref)*>
```

on indique que le corps d'une section est une répétition de 0, 1 ou plusieurs occurrences soit d'une séquence de caractères quelconques (sauf '<' et '&') ou d'un élément bibref, c'est-à-dire une référence bibliographique.

Chaque élément peut avoir 0, un ou plusieurs attributs. La déclaration d'attributs fait intervenir le nom de l'élément auquel l'attribut est rattaché, le nom de l'attribut, le type des valeurs de l'attribut et une indication de valeur par défaut. Les attributs associés à un élément se regroupent dans une liste introduite par la balise ATTLIST :

<!ATTLIST *nom-élément nom-attribut type-attribut défaut* >

Le nom d'élément doit référer à un élément existant dans la DTD. Le nom d'attribut doit respecter les mêmes règles syntaxiques que les noms d'éléments. Le type d'attribut est choisi, entre autres, parmi les types suivants :

CDATA : chaîne de caractères,

ID ou IDREF : renvois à l'intérieur du document,

ENTITY ou ENTITIES : entités externes à XML,

NMTOKEN ou NMTOKENS : identificateurs ou valeurs symboliques,

NOTATION : données interprétables par un outil externe à XML (images, sons, etc.),

liste de choix : de la forme (A|B|C) où A, B et C sont des valeurs symboliques parmi lesquelles la valeur de l'attribut doit être choisie.

L'indication de défaut est choisie parmi les suivantes :

***valeur par défaut* :** une valeur par défaut donnée à l'attribut s'il n'apparaît pas dans la balise,

#REQUIRED : valeur de l'attribut toujours requise,

#IMPLIED : valeur de l'attribut facultative,

#FIXED *valeur* : attribut de valeur constante donnée.

Exemple 5.4 Les déclarations d'éléments suivantes (suivant le prologue) illustrent les différents formes de modèles :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ELEMENT chapitre (titre, intro, (titre-section, corps-section)+, bibel*)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT intro (#PCDATA)>
<!ELEMENT titre-section (#PCDATA)>
<!ELEMENT corps-section (#PCDATA | bibref)*>
<!ELEMENT bibref EMPTY>
<!ATTLIST bibref ref IDREF #REQUIRED>
<!ELEMENT bibel (#PCDATA)>
<!ATTLIST bibel name ID #REQUIRED>
```

Si cette DTD est contenue dans un fichier `chapitre.dtd`, un document valide selon cette DTD est :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE chapitre SYSTEM 'chapitre.dtd'>
<chapitre>
  <titre>Mon chapitre</titre>
  <intro>Ceci est un exemple de document XML</intro>
  <titre-section>Première section</titre-section>
  <corps-section>
    Voir <bibref ref="REC-XML-19980210" /> pour plus de détails.
  </corps-section>
  <titre-section>Seconde section</titre-section>
  <corps-section>Ceci est un corps de section.</corps-section>
  <bibel name="REC-XML-19980210">Ma référence</bibel>
</chapitre>
```

□

5.3 DTD et grammaires

Le cœur d'une DTD (sans les déclarations d'attributs et les types prédéfinins) est équivalente à une grammaire non-contextuelle. Les opérateurs de multiplicité, empruntés au formalisme EBNF, ne sont que du *sucre syntaxique* pouvant être remplacés par des règles de grammaires supplémentaires (qui changerait un peut la forme de la structure du document mais pas sa signification).

Par contre, les DTD sont des grammaires dont la finalité n'est pas exactement la même que celle des grammaires non-contextuelles du chapitre précédent. Les grammaires non-contextuelles servent à définir des structures de chaînes faisant partie d'un langage formel (non-contextuel). Les grammaires au sens des DTD sont utilisées pour définir la forme d'une structure de données arborescente, ici l'arborescence du contenu de documents XML. Cette seconde utilisation des grammaires est aussi très courante, et on appelle souvent ce genre de grammaires des *grammaires d'arbres*.

Dans le domaine de la compilation et des langages de programmation, on utilise les grammaires d'arbres pour définir ce que l'on appelle des arbres de syntaxe abstraite, c'est-à-dire

un arbre qui représente toute la structure et le contenu d'un programme, mais dans lequel tous les aspects syntaxiques externes du langage, comme le choix des mots-clés et les signes de ponctuation (points-virgules, virgules, voire parenthèses) sont omis car devenus inutiles une fois que le programme a été analysé et reconnu conforme à la syntaxe du langage.

Il est intéressant de noter que les propriétés des grammaires non-contextuelles se retrouvent dans le grammaires d'arbres. Nous verrons des applications de cela avec les grammaires attribuées aux chapitres 7 et 8. Cette correspondance justifie le propos de la présente section, où nous allons maintenant voir comment transposer une grammaire non-contextuelle en DTD et comment écrire une phrase du langage reconnu par la grammaire sous la forme d'un document XML valide par rapport à la DTD précédente.

Étudier cette transposition peut paraître un pur exercice de style. De fait ce n'est pas tant la possibilité de transposer une grammaire en une DTD et une phrase dans un document XML qui nous intéressera, mais plutôt l'équivalence entre grammaires/phrases et DTD/document qui va nous permettre d'utiliser les propriétés de l'une dans le contexte de l'autre. Nous en verrons une illustration importante au chapitre 8 lorsque nous utiliserons les propriétés des grammaires attribuées (chapitre 7) pour comprendre les implications sur la performance des transformations de documents XML écrites en XSLT.

D'une grammaire à une DTD

La transformation d'une grammaire non-contextuelle conforme à ce que nous avons vu au chapitre précédent vers une DTD est assez simple. Il suffit d'appliquer les règles suivantes :

1. L'axiome de la grammaire devient le nom du type du document défini par la DTD.
2. Pour chaque non-terminal de la grammaire, introduire un type d'éléments dans la DTD.
3. Pour chaque terminal de la grammaire, introduire un type d'éléments toujours vide dont les éléments vont représenter une occurence de ce terminal.
4. Pour chaque règle de production, utiliser la séquence d'éléments et l'alternative pour traduire la partie gauche des règles de production en définition de type d'élements dans la DTD.

C'est un processus systématique, qui peut être un peu fastidieux, mais sans jamais par contre être bien compliqué.

Exemple 5.5 Peano a donné une définition logique bien connue pour les entiers naturels et les opérations comme l'addition. Dans cette définition, tout entier peut être écrit soit comme étant la constante 0, soit comme le successeur d'un entier n ce qui s'écrit $s(n)$. En terme de règles de grammaire, ceci peut s'écrire :

$$N \rightarrow \mathsf{s}\,N \mid \mathsf{0}$$

Étant donné deux entiers m et n, l'arithmétique de Peano définit l'addition par deux régles de réécriture qui disent respectivement que l'addition d'un entier différent de 0, c'est-à-dire de la forme $s(i)$ avec une entier n peut se réécrire en l'addition de i et $s(n)$, et que l'addition de 0 avec un entier n se réécrit en n. Nous reviendrons au chapitre 7 sur la prise en compte de la réécriture pour faire l'addition. Pour l'instant, en terme de grammaire, une expression

est soit l'addition de deux entiers soit un entier (une constante) :

$$E \quad \rightarrow \quad N \mid \texttt{add}\, N\, N$$

Selon les règles édictées ci-haut, le non-terminal E devient le type de document. Deux élements correspondent aux non-terminaux E et N. Dans les deux cas, il y a deux alternatives de règles de production qui se retrouvent dans les définitions d'éléments, parenthésées comme il se doit pour respecter la priorité des opérateurs dans la DTD. Enfin, trois éléments vides viennent définir les trois terminaux `add`, `s` et `zero` :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ELEMENT e (n | (add, n, n)) >
<!ELEMENT n ((s, n) | zero) >
<!ELEMENT add EMPTY >
<!ELEMENT s EMPTY >
<!ELEMENT zero EMPTY >
```

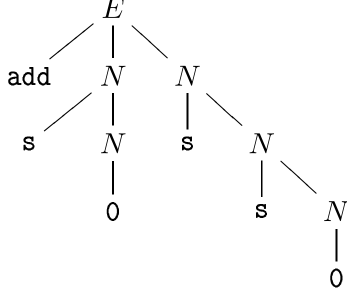
□

D'une phrase acceptée par la grammaire à un document XML

Voyons maintenant comment prendre une phrase acceptée par la grammaire pour en faire un document XML valide par rapport à la DTD obtenue par les règles précédentes. La DTD exprime la forme de l'arbre d'analyse correspondant à la phrase. Il est ainsi plus facile d'obtenir le document XML en passant d'abord par l'arbre d'analyse de la phrase. Une fois l'arbre d'analyse rendu explicite, le document XML pourra être obtenue en faisant correspondre à chaque nœud dans cet arbre un élément dans le document XML. Voici donc comment réaliser cette correspondance :

1. Construire l'arbre d'analyse de la phrase en utilisant la grammaire.
2. Écrire le prologue du document XML en mettant bien comme DTD celle qui a été préalablement définie pour la grammaire considérée.
3. Traverser l'arbre d'analyse et pour chaque nœud rencontré, créer un élément du type correspondant au terminal ou au non-terminal apparaîssant dans le nœud.
4. Lorsque le nœud rencontré fait apparaître un non-terminal, on obtient le contenu de l'élément correspondant dans le document en parcourant chacun des nœuds fils et en faisant la correspondance pour chacun de ceux-ci.

Exemple 5.6 Considérons la phrase $add\, s\, 0\, s\, s\, 0$ acceptée par la grammaire de l'arithmétique de Peano de l'exemple précédent. L'arbre d'analyse correspondant est :



Outre le prologue du document XML, on traduit la phrase en un arbre d'éléments. En parcourant l'arbre d'analyse, l'élément racine fait apparaître le non-terminal E auquel on fait correspondre un élement de type `e`. Le premier fils de ce nœud est le terminal `add` auquel on fait correspondre l'élément vide `<add/>`. Les deux autres fils font apparaître le non-terminal N auquel à chaque fois on fait correspondre un élément de type `n`. En continuant la correspondance, on obtient le document suivantoù on retrouve bien la structure de l'arbre d'analyse précédent :

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='no' ?>
<!DOCTYPE e SYSTEM 'peano.dtd'>
<e>
  <add/>
  <n>
    <s/>
    <n><zero/></n>
  </n>
  <n>
    <s/>
    <n>
      <s/>
      <n><zero/></n>
    </n>
  </n>
</e>
```

□

5.4 Les schémas XML

à compléter

Chapitre 6

Construction des analyseurs pour grammaires non-contextuelles

Comme pour les langages réguliers, il est important de disposer de moyens pour construire des accepteurs pour le langages non-contextuels, car c'est eux qui, d'un point de vue pratique, permettent de vérifier l'appartenance d'une chaîne (ou phrase) à un certain langage non-contextuel. Comme les expressions régulières, les grammaires non-contextuelles offrent un bon moyen pour définir les langages non-contextuels de manière intensionnelle. À partir d'une grammaire, on peut chercher à prouver des propriétés sur le langage qu'elle définit. Mais l'accepteur reste l'outil de base pour la plupart des applications pratiques. Ce chapitre étudie ces différents aspects de la constuction des accepteurs ou analyseurs pour les grammaires non-contextuelles.

6.1 Analyse des langages non-contextuels

Pour ce qui concerne les grammaires non-contextuelles, un vocabulaire adapté est apparu pour parler de reconnaissance et de langages. Là où pour les expressions régulières on parle de chaînes et d'accepteurs, on parle ici plus volontiers de phrases et d'analyseurs. Dans les deux cas, on peut penser que cela tient à la proximité avec le concept de grammaire dans les langages naturels (français, anglais, ...) qui se réfèrent à la construction des phrases. Dans le second cas, cela tient aussi probablement au fait que la principale application des grammaires non-contextuelles est la définition de la syntaxe des langages de programmation où le processus par lequel on vérifie l'exactitude syntaxique d'un programme est appelée analyse syntaxique, d'où le terme analyseur.

L'analyse d'un langage non-contextuel est une procédure qui, pour une grammaire non-contextuelle $G = (V, T, P, S)$, vérifie si une phrase $\omega \in T^*$ appartient à $L(G)$ ou non. Un algorithme simple, dû à Cocke, Younger et Kasami, permet de faire cette vérification pour n'importe quelle grammaire non-contextuelle G. Cet algorithme est basé sur la technique de programmation dynamique, mieux connue en recherche opérationnelle où elle permet de résoudre des problèmes de décision séquentiels. Avant d'aborder l'algorithme définissons la forme normale de Chomsky :

> **Forme normale de Chomsky :** une grammaire $G = (V, T, P, S)$ est dite en forme normale de Chomsky si et seulement si toutes les productions $p \in P$ sont de la forme $A \rightarrow BC$ ou $A \rightarrow \mathtt{a}$ où $A, B, C \in V$ et $\mathtt{a} \in T$.

Soit une grammaire $G = (V, T, P, S)$ et une phrase $x \in T^*$, définissions V_{ij} comme l'ensemble des non-terminaux $A \in V$ tels que $A \stackrel{*}{\Rightarrow} x_{ij}$, alors l'algorithme suivant calcule les V_{ij} pour $1 \leq i, j \leq |x|$:

```
1.  pour i := 1 jusqu'à n faire
2.      V_ij = {A | A → a est une production et que le ième symbole de x est a }
    fin
3.  pour j := 2 jusqu'à n faire
4.      pour i := 1 jusqu'à n − j + 1 faire
5.          V_ij = ∅
6.          pour k := 1 jusqu'à j − 1 faire
7.              V_ij = V_ij ∪ {A | A → BC est une production et B ∈ V_ik et C ∈ V_i+k,j−k }
            fin
        fin
    fin
```

FIG. 6.1 – Algorithme de Cocke, Younger et Kasami.

Théorème 6.1 *(forme normale de Chomsky) Tout langage non-contextuel n'incluant pas ϵ est généré par une grammaire en forme normale de Chomsky.*

L'algorithme CYK (pour Cocke, Younger et Kasami) prend en entrée une grammaire G en forme normale de Chomsky et une phrase x telle que $|x| > 0$, et il détermine pour tout $1 \leq i, j \leq |x|$ et pour tout non-terminal A s'il existe une dérivation $A \stackrel{*}{\Rightarrow} x_{ij}$ où x_{ij} est la sous-phrase de x commençant à l'indice i et se terminant à l'indice j.

L'algorithme procède par induction sur j. Pour $j = 1$, $A \stackrel{*}{\Rightarrow} x_{ij}$ ssi $A \rightarrow x_{ij}$ est une production de la grammaire, car x_{ij} doit avoir pour longueur 1. Pour $j > 1$, $A \stackrel{*}{\Rightarrow} x_{ij}$ ssi il existe une production ABC et un indice $i \leq k > j$ tels que B dérive les k premiers symboles de x et C dérive les $j - k$ derniers symboles de x. Puisque k et $j - k$ sont strictement inférieurs à j, il suffit de vérifier les dérivations en ordre croissant de longueur de sous-phrases.

La figure 6.1 présente l'algorithme CYK. Le grand avantage de cet algorithme est sa complétude par rapport à la classe des langages non-contextuels. Son gros inconvénient est sa consommation de ressources. D'une part, en pire cas, on peut prendre jusqu'à $O(n^3)$ étapes de calcul pour décider si x telle que $|x| = n$ appartient à $L(G)$ ou non. D'autre part, l'ensemble de la phrase dans son entier est accédée, d'où une grande consommation d'espace si la phrase est de grande taille.

Même si CYK est dans $O(n^2)$ pour les grammaires non-ambigües, et souvent linéraire dans beaucoup de cas, il n'est pas considéré comme pratiquement utilisable. D'autres algorithmes ont donc été trouvés qui sont moins gourmands en temps et en espace. L'inconvénient de ces algorithmes vient de ce qu'ils ne sont plus complets au sens où seule une partie des grammaires non-contextuelles peuvent être traitées par chacun de ces algorithmes. Malgré tout, ces classes de grammaires s'avèrent souvent suffisantes en pratique.

Comment sont construits ces algorithmes d'analyse plus performants ? Essentiellement selon deux axes. Nous savons que pour analyser une phrase, il suffit de trouver une dérivation, ou de manière équivalente un arbre d'analyse. Si on considère la dérivation, nous avons mentionné au chapitre 4 que deux grandes stratégies sont possibles pour produire une dérivation :

- On peut partir de l'axiome et essayer de construire la dérivation de gauche à droite en remplaçant à chaque étape un non-terminal dans la protophrase courante par l'une des partie droite de ses règles de production.
- On peut aussi partir de la phrase à dériver et essayer de remonter vers l'axiome en utilisant en quelque sorte des *anti-dérivations* où on remplace une séquence de symboles terminaux ou non-terminaux dans la protophrase courante par un non-terminal pour obtenir la protophrase précédente.

Dans le cas de la construction de gauche à droite, cela revient à construire l'arbre de dérivation de la racine vers les feuilles, c'est-à-dire de du haut vers le bas, d'où le nom d'*analyse descendante*. Dans le second cas, il s'agit plutôt de constuire l'arbre des feuilles vers la racine, d'où le nom d'*analyse ascendante*. Nous allons voir plus précisément ces deux approches dans les prochaines sections.

6.2 Analyse descendante

Comme son nom l'indique, l'analyse descendante consiste à construire la dérivation de la phrase en partant de l'axiome et en allant vers la phrase. Elle s'applique à des grammaires qui ne sont pas récursives à gauche. Considérons d'abord l'idée générale de l'approche descendante, avant d'en voir les différentes variantes.

6.2.1 Idée générale de l'analyse descendante

L'idée générale de cette approche consiste à partir de l'axiome à considérer la phrase pour se demander comment dériver les premiers symboles de la phrase en remplaçant l'axiome par l'une des parties droites de ses règles de production.

Une fois la première dérivation faite, de deux choses l'une : soit on a introduit une séquence de terminaux au début de la nouvelle protophrase courante, soit la nouvelle protophrase courante commence par un non-terminal. Si des non-terminaux sont apparus au début de la protophrase courante, on les apparie avec les terminaux débutant la phrase et ce jusqu'à ce qu'on rencontre un non-terminal dans la protophrase courante. Lorsqu'un non-terminal est atteint dans la protophrase courante, on se demande à nouveau comment générer les prochains terminaux dans la phrase en le remplaçant par l'une des parties droites de ses règles de production. Et ainsi de suite, jusqu'à ce que la phrase ait pu être produite, ou jusqu'à ce que l'on rencontre un symbole terminal dans la protophrase courant qui ne s'apparie pas avec le prochain terminal dans la phrase.

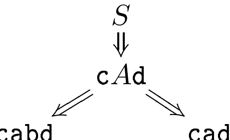
Si on arrive à un échec sur l'appariement du prochain terminal dans la phrase à partir du prochain terminal dans la protophrase, alors de deux choses l'une : soit il faut remettre en cause l'un des choix de règle de production faits depuis le début de l'analyse pour essayer de dériver autrement la phrase, soit il n'y a pas d'autres choix possibles (et on en est sûr) auquel cas, on peut déclarer que la phrase n'appartient pas au langage défini par la grammaire.

Le processus que nous venons décrire correspond en fait à une fouille dans un arbre pour trouver la dérivation à gauche de la phrase à analyser. Un nœud dans cette arbre de fouille correspond à une protophrase gauche alors qu'un arc correspond à une dérivation à gauche par l'une des règles de production.

Exemple 6.1 Soit la grammaire :

$$\begin{array}{lcl} S & \rightarrow & \mathtt{c}A\mathtt{d} \\ A & \rightarrow & \mathtt{ab} \mid \mathtt{a} \end{array}$$

Pour cette grammaire très simple, l'arbre de fouille est fini ; il a la forme suivante :

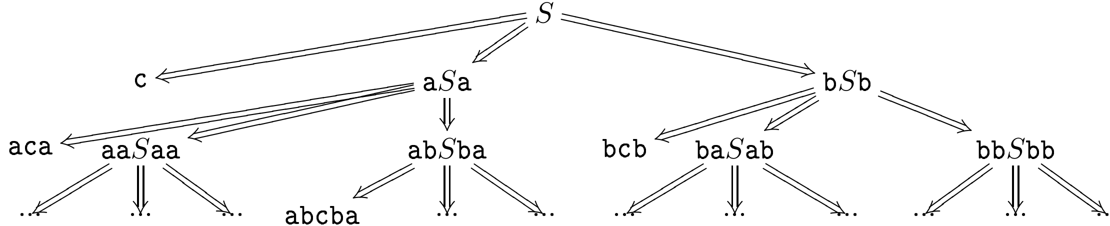


Pourtant, lorsqu'on fouille cet arbre pour analyser la phrase cad, la première dérivation est bien $S \Rightarrow \mathtt{c}A\mathtt{d}$, mais la suivante pose des difficultés. Le symbole 'c' introduit en première position correspond bien au premier symbole de la phrase à reconnaître, mais le second symbole, 'a', apparaît à la fois dans la branche de gauche et dans la branche de droite du nœud courant de l'abre de fouille. En considérant d'abord le nœud de gauche, on reconnaît bien le 'a', mais lorsqu'on passe au symbole suivant 'd' dans la phrase à analyser, on échoue à reconnaître le 'b' introduit dans la protophrase courante. Il faut donc revenir à la protophrase précédente et tenter la deuxième dérivation possible, qui donne la protophrase cad qui correspond bien à la phrase à analyser. □

Exemple 6.2 La grammaire suivante engendre le langage $L = \{\omega \mathtt{c} \omega^R \,|\, \omega \in (\mathtt{a}|\mathtt{b})^*\}$:

$$S \quad \rightarrow \quad \mathtt{c} \mid \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b}$$

L'rbre de fouille engendré à partir de cette grammaire est :



Si on veut reconnaître la phrase abcba, on doit d'abord choisir la dérivation $S \Rightarrow \mathtt{a}S\mathtt{a}$, puisque c'est la seul à cette étape qui permet d'introduire un symbole 'a' en première position de la protophrase courante. Ce symbole reconnu, le prochain symbole à reconnaître étant 'b', on choisit la dérivation $\mathtt{a}S\mathtt{a} \Rightarrow \mathtt{ab}S\mathtt{ba}$. Enfin, après la reconnaissance du 'b', le prochain symbole à reconnaître étant 'c', on choisit la dérivation suivante comme $\mathtt{ab}S\mathtt{ba} \Rightarrow \mathtt{abcba}$. □

De cette explication générale et des exemples précédents, on confirme que l'analyse descendante produit en fait une dérivation à gauche, dans la mesure où en avançant en parallèle dans la protophrase courante et dans la phrase à analyser, on réduit toujours le non-terminal le plus à gauche dans la protophrase. Cette caractéristique est commune à toutes les variantes de l'analyse descendante.

Parmi les variantes de l'analyse descendante, l'analyse non-prédictive va remettre en cause les choix des règles faits depuis le début de l'analyse en cas d'échec de l'appariement du prochain terminal à reconnaître, par exemple en utilisant le retour arrière, et ce tant qu'il existe d'autres choix possibles. C'est le cas de l'exemple 6.1.

Dans l'analyse prédictive, on va s'intéresser à une classe de grammaires non-contextuelles pour lesquelles il est possible de décider à coup sûr à chaque étape quelle règle de production choisir en regardant simplement les quelques prochains terminaux à produire dans la phrase à analyser. C'est le cas de l'exemple 6.2. Cette classe de grammaire est appelée $LL(k)$, ce qui veut dire qu'elle est analysable en parcourant la phrase à reconnaître de gauche à droite («*left to right scanning of the input*») par dérivation à gauche («*leftmost derivation*») avec k symboles de prévision. Les k symboles de prévision s'interprètent comme le fait qu'à chaque étape de dérivation on peut choisir à coup sûr la prochaine règle de production à utiliser ou déduire l'échec de la dérivation en regardant simplement les k prochains symboles à reconnaître dans la phrase. En pratique, on se limite le plus souvent à des grammaires $LL(1)$.

6.2.2 Descente récursive

La descente récursive est une technique générale de construction d'analyseurs descendants. Elle peut se faire avec ou sans retour arrière, c'est-à-dire en analyse prédictive ou non-prédictive. L'idée générale de construction d'un analyseur par descente récursive consiste à écrire une procédure pour chacun des symboles non-terminaux de la grammaire. Cette procédure est chargée d'appliquer les différentes règles de production pour analyser la phrase.

Appliquer une règle de production est relativement simple. On considère les symboles de la partie droite dans l'ordre de manière à ce que l'application d'une règle de production s'interprète comme la «consommation» dans la phrase de tous les terminaux dérivés à partir de cette règle. Pour chaque symbole terminal `a`, on insère dans la procédure un appel à une procédure `décaler` qui va chercher à apparier le symbole `a` dans la phrase et si cela réussit va passer au symbole suivant dans la phrase (on dit qu'on *décale* ce symbole, ou qu'on le consomme). Pour chaque symbole non-terminal A, on insère un appel à la procédure construite pour pour ce symbole qui, par définition, va consommer tous les terminaux engendrés par ce non-terminal dans la phrase (à partir de la position courante).

On lance l'analyse en appelant la procédure correspondante à l'axiome. On parle de descente parce qu'il s'agit d'une analyse descendante. On dit récursive parce que les procédures s'appellent l'une l'autre, et lorsqu'il y a récursivité dans les règles de production (ce qui est presque toujours le cas), cette récursivité se retrouve dans les appels de l'analyseur.

Descente récursive non-prédictive

Dans un analyseur par descente récursive non-prédictive, comme nous l'avons mentionné, il faut être en mesure de revenir en arrière sur la tentative de dérivation en remettant en cause les choix de règles faits précédemment. Un algorithme général pour faire ce genre de recherche s'appelle est le *retour arrière*. Dans un langage classique comme Java, cela suppose généralement de créer une pile où on note l'état courant de la reconnaissance (règle choisie et position dans la phrase) à chaque fois qu'on choisit une règle, et lorsqu'on arrive à un échec d'appariement, à revenir au contexte en sommet de pile pour faire un autre choix.

Certains langages incluent des fonctionnalités de ce genre. Prolog, par exemple, cherche à démontrer un certain but en utilisant des règles d'inférences, et il utilise nativement le retour arrière pour chercher (de manière astucieuse) parmi toutes les séquences d'application des règles d'inférences, celles qui permettront de démontrer le but. Il est tout à fait possible d'utiliser Prolog pour programmer une descente récursive non-prédictive, et ainsi bénéficier du mécanisme de retour arrière natif de ce langage.

Calcul de la fonction Premier

Premier(X), pour tout symbole de la grammaire X :

- Si X est un terminal, alors Premier(X) $= \{X\}$.
- Si $X \rightarrow \epsilon$ est une production, ajouter ϵ à Premier(X).
- Si X est un non-terminal et que $X \rightarrow Y_1Y_2...Y_k$ est une production, mettre a dans Premier(X) s'il existe i, $0 \leq i < k$ tel que $\epsilon \in$ Premier(Y_j) pour $0 \leq j < i$ et $a \in$ Premier(Y_i).

Premier($X_1X_2...X_n$), pour une séquence de symboles X_i de la grammaire :

1. $\forall a \in$ Premier(X_1) et a est un terminal, ajouter a dans Premier($X_1X_2...X_n$).
2. $i := 1$
3. **tant que** $\epsilon \in$ Premier(X_i) et $i < n$ **faire**
4. $\quad$ $\forall a \in$ Premier(X_{i+1}), ajouter a dans Premier($X_1X_2...X_n$).
5. $\quad$ $i := i + 1$

   fin
6. **si** $\epsilon \in$ Premier(X_i), $1 \leq i \leq n$ **alors**
7. $\quad$ ajouter ϵ dans Premier($X_1X_2...X_n$).

Calcul de la fonction Suivant

Suivant(A) pour tous les non-terminaux A de la grammaire :

- Mettre \$ dans Suivant(S), où S est l'axiome de la grammaire et \$ le marqueur de fin de la chaîne d'entrée.
- S'il y a une production $A \rightarrow \alpha B\beta$, le contenu de Premier($\beta$) excepté ϵ est ajouté à Suivant(B).
- S'il existe une production $A \rightarrow \alpha B$ ou une production $A \rightarrow \alpha B\beta$ telle que Premier(β) contient ϵ (c'est-à-dire $\beta \overset{*}{\Rightarrow} \epsilon$), les éléments de Suivant(A) sont ajoutés à Suivant(B).

Fig. 6.2 – Calcul des fonctions Premier et Suivant.

L'inconvénient de l'approche non-prédictive et la relative inefficacité de la recherche par retour arrière. Elle n'est donc appliquée que dans des cas où les chaînes à analyser sont relativement petites et où on cherche à programmer très rapidement des analyseurs qui fonctionnement pour des grammaires qui ne sont pas très bien conditionnées pour les autres méthodes.

Descente récursive prédictive

Pour programmer une descente récursive prédictive, il faut d'abord travailler sur la grammaire de manière à déterminer les critères de choix des règles de production à chaque fois qu'on doit remplacer un non-terminal par l'une de ses parties droites de règles de production. Il s'agit donc de savoir quel est la séquence des premiers symboles terminaux qui peut être engendrée par chaque partie droite de règle de production. Selon le cas, on peut vouloir regarder un, deux, trois ou plus symboles de prédiction. En pratique, on se contente souvent de traiter les grammaires qui peuvent être analysée prédictivement avec un seul symbole de prévision.

Pour une grammaire G qui n'est pas récursive à gauche, un analyseur par descente récursive prédictive se construit donc en calculant d'abord les fonctions Premier et Suivant tel que cela est décrit à la figure 6.2. La fonction Premier, lorsqu'appliquée à une séquence de

symboles, retourne le premier symbole terminal de la phrase engendrée par cette séquence de symboles. La fonction SUIVANT, lorsqu'appliquée à un symbole non-terminal, retourne tous les terminaux qui peuvent suivre ce symbole dans une protophrase engendrée par la grammaire.

Une fois ces fonctions calculées, on crée des procédures de reconnaissance pour chacun des symbole non-terminaux de G. Les procédures de reconnaissance utilisent un tampon d'entrée ω et une pile de symboles terminaux et non-terminaux P gérés par les trois opérations suivantes :

1. `décaler(`a`)` : si le prochain terminal à reconnaître dans ω est a, alors ce symbole a est retiré de ω et empilé sur P.
2. `réduire(`n`, `A`)` : dépile n symboles du sommet de P et empile le non-terminal A à leur place ; la production appliquée réduit donc A vers la séquence de symboles dépilés (la partie droite de la production étant constituée des symboles dépilés dans l'ordre de dépilement).
3. `prochain(`a`)` : retourne vrai si le prochain terminal dans ω est égal à a et faux sinon.

Ces procédures définies, il faut traiter chaque non-terminal de la grammaire pour programmer les procédures de reconnaissance correspondantes. Pour chaque production $A \to \alpha$ de la grammaire, on doit calculer les symboles de prévision associés à cette production :

$$\text{PRÉVISION}(A \to \alpha) = \begin{cases} \text{PREMIER}(\alpha) & \text{si } \epsilon \notin \text{PREMIER}(\alpha) \\ (\text{PREMIER}(\alpha) \backslash \{\epsilon\}) \cup \text{SUIVANT}(A) & \text{sinon} \end{cases}$$

La grammaire *ne se prête pas* à une analyse descendante prédictive s'il existe au moins deux productions $A \to \alpha_1$ et $A \to \alpha_2$ telles que $\text{PRÉVISION}(A \to \alpha_1) \cap \text{PRÉVISION}(A \to \alpha_2) \neq \emptyset$. Si la grammaire se prête à une analyse descendante prédictive, pour chaque non-terminal A dont les productions sont $A \to \alpha_1|...|\alpha_n$, écrire une procédure de reconnaissance de la forme suivante :

procédure reconnait$A$()
début
  si $\exists$c $\in$ PRÉVISION($A \to \alpha_1$) tel que `prochain(c)` **alors**
    appliquer la production $A \to \alpha_1$
  sinon si $\exists$c $\in$ PRÉVISION($A \to \alpha_2$) tel que `prochain(c)` **alors**
    appliquer la production $A \to \alpha_2$
  ...
  sinon si $\exists$c $\in$ PRÉVISION($A \to \alpha_n$) tel que `prochain(c)` **alors**
    appliquer la production $A \to \alpha_n$
  sinon
    déclencher une erreur de syntaxe.
fin

Pour appliquer une production de la forme $A \to X_1...X_k$, il faut produire une séquence d'énoncés $e_1, ..., e_k$ de la forme :

$$e_i = \begin{cases} \texttt{décaler(}X_i\texttt{) ;} & \text{si } X_i \text{ est un terminal} \\ \texttt{reconnait}X_i\texttt{() ;} & \text{sinon} \end{cases}$$

puis on termine par un énoncé `réduire(`k`, `A`) ;` pour marquer la reconnaissance de la partie droite de la production (si la production est de la forme $A \to \epsilon$, alors l'application se résume à `réduire(0, `A`) ;`).

L'analyse par descente récursive prédictive est lancée en appelant la procédure de reconnaissance de l'axiome dans un contexte où la pile P est vide et en considérant le premier

symbole de ω comme le prochain symbole à reconnaître. Cette analyse réussit si elle parvient au bout de ω en laissant l'axiome comme seul symbole dans la pile.

Exemple 6.3 Considérons la grammaire suivante :

$$
\begin{array}{lcl}
E & \rightarrow & E\texttt{+}T \mid E\texttt{-}T \mid T \\
T & \rightarrow & T\texttt{*}F \mid T\texttt{/}F \mid F \\
F & \rightarrow & F\texttt{\^{}}P \mid P \\
P & \rightarrow & \texttt{(}E\texttt{)} \mid N \\
N & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

D'abord, on note que cette grammaire est récursive à gauche. Il faut éliminer ses récursivités à gauche, ce qui nous donne la grammaire équivalente suivante :

$$
\begin{array}{lcl}
E & \rightarrow & TX \\
X & \rightarrow & \texttt{+}TX \mid \texttt{-}TX \mid \epsilon \\
T & \rightarrow & FY \\
Y & \rightarrow & \texttt{*}FY \mid \texttt{/}FY \mid \epsilon \\
F & \rightarrow & PZ \\
Z & \rightarrow & \texttt{\^{}}PZ \mid \epsilon \\
P & \rightarrow & \texttt{(}E\texttt{)} \mid N \\
N & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

Ensuite, calculons la fonction Premier pour tous les symboles de la grammaire. Pour les terminaux, Premier retourne le symbole lui-même. Pour les non-terminaux, on obtient :

$$
\begin{array}{rcl}
\textsc{Premier}(E) & = & \textsc{Premier}(T) = \{\texttt{(}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\textsc{Premier}(X) & = & \{\texttt{+}, \texttt{-}, \epsilon\} \\
\textsc{Premier}(T) & = & \textsc{Premier}(F) = \{\texttt{(}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\textsc{Premier}(Y) & = & \{\texttt{*}, \texttt{/}, \epsilon\} \\
\textsc{Premier}(F) & = & \textsc{Premier}(P) = \{\texttt{(}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\textsc{Premier}(Z) & = & \{\texttt{\^{}}, \epsilon\} \\
\textsc{Premier}(P) & = & \{\texttt{(}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\textsc{Premier}(N) & = & \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
\end{array}
$$

On passe ensuite au calcul de la fonction Suivant pour tous les non-terminaux de la grammaire :

$$
\begin{array}{rcl}
\textsc{Suivant}(E) & = & \{\$\} \cup \{\texttt{)}\} = \{\$, \texttt{)}\} \\
\textsc{Suivant}(X) & = & \textsc{Suivant}(E) \cup \textsc{Suivant}(X) = \{\$, \texttt{)}\} \\
\textsc{Suivant}(T) & = & \textsc{Premier}(X)\backslash\{\epsilon\} \cup \textsc{Suivant}(E) \cup \textsc{Suivant}(X) = \{\texttt{+}, \texttt{-}, \$, \texttt{)}\} \\
\textsc{Suivant}(Y) & = & \textsc{Suivant}(T) \cup \textsc{Suivant}(Y) = \{\texttt{+}, \texttt{-}, \$, \texttt{)}\} \\
\textsc{Suivant}(F) & = & \textsc{Premier}(Y)\backslash\{\epsilon\} \cup \textsc{Suivant}(T) \cup \textsc{Suivant}(Y) = \{\texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \$, \texttt{)}\} \\
\textsc{Suivant}(Z) & = & \textsc{Suivant}(F) \cup \textsc{Suivant}(Z) = \{\texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \$, \texttt{)}\} \\
\textsc{Suivant}(P) & = & \textsc{Premier}(Z)\backslash\{\epsilon\} \cup \textsc{Suivant}(F) \cup \textsc{Suivant}(Z) = \{\texttt{\^{}}, \texttt{*}, \texttt{/}, \texttt{+}, \texttt{-}, \$, \texttt{)}\}
\end{array}
$$

```
import java.io.* ;

public class                        DescenteRecursive
{
    /** Représentation textuelle du programme                            */
    private static      char[]          programme ;
    /** Position courante atteinte dans le programme par l'analyse      */
    private static      int             positionCourante ;
    /** Nombre total de symboles dans le texte du programme             */
    private static      int             nombreSymboles ;

    private static void         decale(char prochainCar)
    {
        if ( programme[positionCourante] == prochainCar ) {
            positionCourante++ ;
        } else {
            System.out.println("Erreur de syntaxe") ;
        }
    }

    private static boolean      verifieProchain (char prochainCar)
    {
        boolean         ret ;

        if ( positionCourante < nombreSymboles ) {
            ret = (programme[positionCourante] == prochainCar) ;
        } else if ( positionCourante == nombreSymboles && prochainCar == '$') {
            ret = true ;
        } else {
            ret = false ;
        }
        return ret ;
    }

    private static void         reduit (int nSymbolesRed, char partieGauche)
    {
        int     posSuivante,  i ;

        if ( nSymbolesRed == 0 ) {
            for ( i = nombreSymboles ; i > positionCourante ; i-- ) {
                programme[i] = programme[i-1] ;
            } // end of for ()
            programme[positionCourante++] = partieGauche ;
            nombreSymboles++ ;
        } else {
            posSuivante = positionCourante ;
            positionCourante -= nSymbolesRed ;   // dépile les symboles
                                                 // réempile la partie gauche
            programme[positionCourante++] = partieGauche ;
                                                 // décale vers la gauche les
                                                 // caractères suivants
            for ( i = 0 ; posSuivante + i < nombreSymboles ; i++ ) {
                programme[positionCourante + i] = programme[posSuivante + i] ;
            }
            for ( i = 1 ; i < nSymbolesRed ; i++ ) {
                programme[nombreSymboles - i] = ' ' ;
            }
            nombreSymboles = (nombreSymboles - nSymbolesRed) + 1 ;
        }
    }

} // --------------------------------------------- classe DescenteRecursive
```

FIG. 6.3 – Descente récursive prédictive en Java.

Ces fonctions calculées, on trouvera à la figure 6.3 donne les définitions relatives à la gestion de la phrase et de la pile. On doit enfin passer à la programmation des procédures de reconnaissance. Pour la procédure de reconnaissance de N, le décalage étant spécifique à un terminal donné, la procédure suivante détecte la règle de production à appliquer en vérifier le premier symbole dérivable de chacune des parties droites des règles de production (on a factorisé la réduction à la fin de la procédure) :

```
private static void         reconnaitN()
{
    if       ( verifieProchain('0') ) { decale('0') ;
    } else if ( verifieProchain('1') ) { decale('1') ;
    } else if ( verifieProchain('2') ) { decale('2') ;
    } else if ( verifieProchain('3') ) { decale('3') ;
    } else if ( verifieProchain('4') ) { decale('4') ;
    } else if ( verifieProchain('5') ) { decale('5') ;
    } else if ( verifieProchain('6') ) { decale('6') ;
    } else if ( verifieProchain('7') ) { decale('7') ;
    } else if ( verifieProchain('8') ) { decale('8') ;
    } else if ( verifieProchain('9') ) { decale('9') ;
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    } // end of else
    reduit(1, 'N') ;
}
```

Pour le non-terminal P, nous avons deux règles de production. La première a comme symboles de prévision $\text{PREMIER}((E)) = \{(\}$. Pour la seconde règle de production, on doit reconnaître un nombre d'où $\text{PREMIER}(N) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. On obtient la procédure de reconnaissance :

```
private static void         reconnaitP ()
{
    if ( verifieProchain('(') ) {
        decale('(') ;
        reconnaitE() ;
        decale(')') ;
        reduit(3, 'P') ;
    } else if ( verifieProchain('0') || verifieProchain('1') ||
                verifieProchain('2') || verifieProchain('3') ||
                verifieProchain('4') || verifieProchain('5') ||
                verifieProchain('6') || verifieProchain('7') ||
                verifieProchain('8') || verifieProchain('9') ) {
        reconnaitN() ;
        reduit(1, 'P') ;
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    }
}
```

Pour le non-terminal Z, on a deux règles de production. Pour la première, on a comme symboles de prévision $\text{PREMIER}(\hat{}PZ) = \{\hat{}\}$. Pour la seconde règle, la partie droite est ϵ ; cette règle, ne consommant pas de symboles, ne s'appliquent que si le prochain symbole à reconnaître fait partie des suivants de Z, donc $\{*, /, +, -, \$,)\}$. On obtient donc :

```
private static void         reconnaitZ()
{
    if ( verifieProchain('^') ) {
        decale('^') ;
        reconnaitP() ;
        reconnaitZ() ;
        reduit(3, 'Z') ;
    } else if ( verifieProchain('*') || verifieProchain('/') ||
                verifieProchain('+') || verifieProchain('-') ||
                verifieProchain('$') || verifieProchain(')') ) {
        reduit(0, 'Z') ;
```

```
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    }
}
```

La reconnaissance d'un facteur fait intervenir une seule règle de production qui s'applique si le prochain symbole à reconnaître fait partie de PREMIER$(PZ) = \{(, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. On a donc :

```
private static void        reconnaitF ()
{
    if ( verifieProchain('(') || verifieProchain('0') ||
         verifieProchain('1') || verifieProchain('2') ||
         verifieProchain('3') || verifieProchain('4') ||
         verifieProchain('5') || verifieProchain('6') ||
         verifieProchain('7') || verifieProchain('8') ||
         verifieProchain('9')) {
        reconnaitP() ;
        reconnaitZ() ;
        reduit(2, 'F') ;
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    }
}
```

Pour la reconnaissance d'un Y, on a trois règles de production dont les symboles de prévisions sont respectivement `*`, `/` et les suivants de Y, d'où :

```
private static void        reconnaitY()
{
    if ( verifieProchain('*') ) {
        decale('*') ;
        reconnaitF() ;
        reconnaitY() ;
        reduit(3, 'Y') ;
    } else if ( verifieProchain('/') ) {
        decale('/') ;
        reconnaitF() ;
        reconnaitY() ;
        reduit(3, 'Y') ;
    } else if ( verifieProchain('+') || verifieProchain('-') ||
                verifieProchain('$') || verifieProchain(')') ) {
        reduit(0, 'Y') ;
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    }
}
```

Pour le non-terminal T, il n'y a encore qu'une seule règle dont les symboles de prévision sont les premiers de F, d'où :

```
private static void        reconnaitT()
{
    if ( verifieProchain('(') || verifieProchain('0') ||
         verifieProchain('1') || verifieProchain('2') ||
         verifieProchain('3') || verifieProchain('4') ||
         verifieProchain('5') || verifieProchain('6') ||
         verifieProchain('7') || verifieProchain('8') ||
         verifieProchain('9')) {
        reconnaitF() ;
        reconnaitY() ;
        reduit(2, 'T') ;
    } else {
        System.out.println("Erreur de syntaxe.") ; System.exit(1) ;
    }
}
```

Le non-terminal X se traite de manière similaire à Y et Z. Trois règles de production ont pour symboles de prévision respectifs +, - et les suivants de X, d'où

```
private static void        reconnaitX()
{
    if ( verifieProchain('+') ) {
        decale('+') ;
        reconnaitT() ;
        reconnaitX() ;
        reduit(3, 'X') ;
    } else if ( verifieProchain('-') ) {
        decale('-') ;
        reconnaitT() ;
        reconnaitX() ;
        reduit(3, 'X') ;
    } else if ( verifieProchain('$') || verifieProchain(')') ) {
        reduit(0, 'X') ;
    } else {
        System.out.println("Erreur de syntaxe.") ;
        System.exit(1) ;
    }
}
```

Et enfin, pour le symbole E, il n'y qu'une règle de production qui s'applique si le prochain symbole à reconnaître fait partie des premiers de T, d'où :

```
private static void        reconnaitE()
{
    if ( verifieProchain('(') || verifieProchain('0') ||
         verifieProchain('1') || verifieProchain('2') ||
         verifieProchain('3') || verifieProchain('4') ||
         verifieProchain('5') || verifieProchain('6') ||
         verifieProchain('7') || verifieProchain('8') ||
         verifieProchain('9')) {
        reconnaitT() ;
        reconnaitX() ;
        reduit(2, 'E') ;
    } else {
        System.out.println("Erreur de syntaxe.") ;
        System.exit(1) ;
    }
}
```

Maintenant, on peut tracer l'exécution du programme résultant sur une expression (simple, pour ne pas se noyer dans une longue trace) comme 2+3 :

```
> reconnaitE()
  > reconnaitT()
    > reconnaitF()
      > reconnaitP()
        > reconnaitN()
          > decale('2')
          < decale('2')
          Applique N -> 2
        < reconnaitN()
        Applique P -> N
      < reconnaitP()
      > reconnaitZ()
        Applique Z -> <epsilon>
      < reconnaitZ()
      Applique F -> P Z
    < reconnaitF()
    > reconnaitY()
      Applique Y -> <epsilon>
    < reconnaitY()
    Applique T -> F Y
  < reconnaitT()
  > reconnaitX()
    > decale('+')
```

```
    < decale('+')
    > reconnaitT()
      > reconnaitF()
        > reconnaitP()
          > reconnaitN()
            > decale('3')
            < decale('3')
            Applique N -> 3
          < reconnaitN()
          Applique P -> N
        < reconnaitP()
        > reconnaitZ()
          Applique Z -> <epsilon>
        < reconnaitZ()
        Applique F -> P Z
      < reconnaitF()
      > reconnaitY()
        Applique Y -> <epsilon>
      < reconnaitY()
      Applique T -> F Y
    < reconnaitT()
    > reconnaitX()
      Applique X -> <epsilon>
    < reconnaitX()
    Applique X -> + T X
  < reconnaitX()
  Applique E -> T X
< reconnaitE()
```

□

6.2.3 Analyseurs prédictifs non-récursifs de type LL(1)

Les analyseurs par descente récursive prédictive sont relativement faciles et pas trop fastidieux à construire si la grammaire est relativement petite. On peut cependant leur reprocher une utilisation peu efficace du temps et de l'espace à cause des nombreux appels de méthodes (ce que l'on peut constater dans la trace de l'exemple précédent) qui nécessitent à la fois du temps et consomment passablement d'espace dans la pile d'exécution du programme. En suivant l'un des plus vieux théorèmes de l'informatique, à savoir que tout ce qui peut se faire récursivement peut aussi se faire itérativement, il est possible de construire des analyseurs prédictifs non-récursifs qui utilisent l'itération. Le choix des règles de production à appliquer à chaque étape de dérivation se fait en utilisant une table qui, étant donné le prochain symbole à reconnaître dans la phrase et le non-terminal à réduire, retourne la règle de production à utiliser.

Soit G une grammaire dont la table d'analyse est `M` et ω une chaîne à analyser, l'algorithme d'analyse qui produit une dérivation à gauche de ω en se basant sur le schéma de la figure 6.4 se décline de la façon suivante :

1. Positionner le curseur sur le premier symbole de ω.
2. Initialiser la *pile* avec l'axiome en fond de pile.
3. **tant que** la pile n'est pas vide et l'entrée ω n'est pas terminée **faire**
4.   Soit `Z` le symbole en sommet de pile et `a` le prochain terminal à reconnaître dans ω
5.   **si** $\texttt{Z} = \texttt{a} = \$$ **alors**
6.   fin et succès de l'analyse
7.   **sinon si** $\texttt{Z} = \texttt{a} \neq \$$ `alors`
8.   dépiler `Z` et passer au prochain terminal dans ω
9.   **sinon si** `Z` est un non-terminal `alors`
10.   **si** $\texttt{M[Z, a]} = \{ \texttt{Z} \rightarrow \texttt{UVW} \}$ `alors`

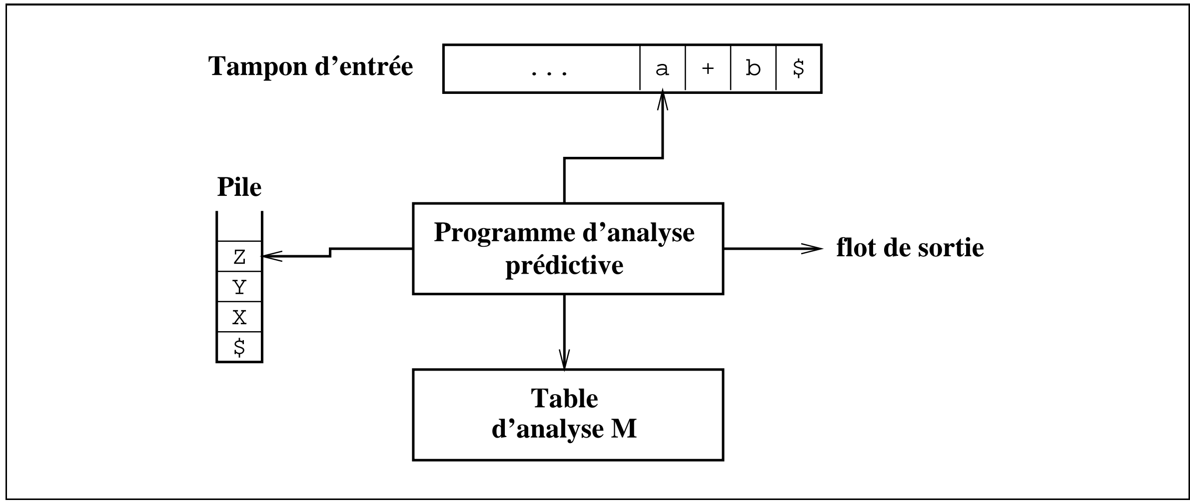


FIG. 6.4 – Schéma du programme d'analyse prédictive

11. dépiler `Z` et empiler `W`, `V` et `U`
12. imprimer `Z` → `UVW` sur le flôt de sortie
 `sinon`
13. erreur de syntaxe
14. **sinon**
15. erreur de syntaxe

fin

L'algorithme de construction de la table d'analyse prédictive itérative est donné à la figure 6.5. Cet algorithme reprend essentiellement les critères de choix des règles de production que nous avons utilisés dans la descente récursive prédictive, en les regroupant dans une table. On reconnaît le cas des symboles parmi les premiers de la partie droite de chaque règle de production, puis celui de l'application des règles produisant la chaîne vide ϵ.

Exemple 6.4 Considérons à nouveau la grammaire de notre exemple précédent :

$$\begin{array}{ll} E \rightarrow TX & F \rightarrow PZ \\ X \rightarrow \texttt{+}TX \mid \texttt{-}TX \mid \epsilon & Z \rightarrow \texttt{\^{}}PZ \mid \epsilon \\ T \rightarrow FY & P \rightarrow \texttt{(}E\texttt{)} \mid N \\ Y \rightarrow \texttt{*}FY \mid \texttt{/}FY \mid \epsilon & \end{array}$$

$$N \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

La table d'analyse obtenue à partir de l'algorithme est :

Soit une grammaire G, l'algorithme produit une table d'analyse prédictive M pour G, si la grammaire G est LL(1). Pour cela, on suppose que les fonctions PREMIER et SUIVANT ont été préalablement calculées.

1. **pour** chaque production $A \rightarrow \alpha$ **faire**
2. **pour** chaque terminal a dans PREMIER(α) **faire**
3. ajouter $A \rightarrow \alpha$ à $M[A, a]$.
   fin
4. **si** $\epsilon \in$ PREMIER(α) **alors**
5. **pour** chaque terminal $b \in$ SUIVANT(A) **faire**
6. ajouter $A \rightarrow \alpha$ à $M[A, b]$.
7. **si** $\$ \in$ SUIVANT(A) **alors**
8. ajouter $A \rightarrow \alpha$ à $M[A, \$]$.
   fin
   fin
9. Faire de chaque entrée non-définie de M une erreur.

FIG. 6.5 – Construction des tables d'analyse LL(1)

	+	-	*	/	^
E					
X	$X \rightarrow$ +TX	$X \rightarrow$ -TX			
T					
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	$Y \rightarrow *FY$	$Y \rightarrow /FY$	
F					
Z	$Z \rightarrow \epsilon$	$Z \rightarrow \epsilon$	$Z \rightarrow \epsilon$	$Z \rightarrow \epsilon$	$Z \rightarrow$ ^PZ
P					
N					

	(	)	n	$	
E	$E \rightarrow TX$		$E \rightarrow TX$		
X		$X \rightarrow \epsilon$		$X \rightarrow \epsilon$	
T	$T \rightarrow FY$		$T \rightarrow FY$		
Y		$Y \rightarrow \epsilon$		$Y \rightarrow \epsilon$	
F	$F \rightarrow PZ$		$F \rightarrow PZ$		
Z		$Z \rightarrow \epsilon$		$Z \rightarrow \epsilon$	
P	$P \rightarrow$ (E)		$P \rightarrow N$		
N			$N \rightarrow n$		

La figure 6.6 retrace l'application de l'algorithme d'analyse au cas de l'expression 2+3. La dérivation obtenue est : $E \underset{g}{\Rightarrow} TX \underset{g}{\Rightarrow} FYX \underset{g}{\Rightarrow} PZYX \underset{g}{\Rightarrow} 2ZYX \underset{g}{\Rightarrow} 2YX \underset{g}{\Rightarrow} 2X \underset{g}{\Rightarrow} 2{+}TX \underset{g}{\Rightarrow} 2{+}FYX \underset{g}{\Rightarrow} 2{+}PZYX \underset{g}{\Rightarrow} 2{+}NZYX \underset{g}{\Rightarrow} 2{+}3ZYX \underset{g}{\Rightarrow} 2{+}3YX \underset{g}{\Rightarrow} 2{+}3X \underset{g}{\Rightarrow} 2{+}3$

□

pile	ω	Décision
E	2+3$	Appliquer $E \rightarrow TX$
XT	2+3$	Appliquer $T \rightarrow FY$
XYF	2+3$	Appliquer $F \rightarrow PZ$
$XYZP$	2+3$	Appliquer $P \rightarrow N$
$XYZN$	2+3$	Appliquer $N \rightarrow 2$
$XYZ2$	2+3$	Décaler 2
XYZ	+3$	Appliquer $Z \rightarrow \epsilon$
XY	+3$	Appliquer $Y \rightarrow \epsilon$
X	+3$	Appliquer $X \rightarrow +TX$
$XT+$	+3$	Décaler +
XT	3$	Appliquer $T \rightarrow FY$
XYF	3$	Appliquer $F \rightarrow PZ$
$XYZP$	3$	Appliquer $P \rightarrow N$
$XYZN$	3$	Appliquer $N \rightarrow 3$
$XYZ3$	3$	Décaler 3
XYZ	$	Appliquer $Z \rightarrow \epsilon$
XY	$	Appliquer $Y \rightarrow \epsilon$
X	$	Appliquer $X \rightarrow \epsilon$
	$	Accepter

FIG. 6.6 – Analyse LL(1) non-récursive de l'expression 2+3.

6.3 Analyse ascendante

L'analyse descendante considère la phrase de gauche à droite et cherche à produire une dérivation à gauche en partant de l'axiome pour arriver éventuellement à la phrase entière. Ce faisant, à chaque étape, l'analyse descendante prédictive choisit les règles de production à appliquer en se fondant uniquement sur le prochain (ou les quelques prochains) symbole à reconnaître dans la phrase.

L'analyse ascendante procède de manière diamétralement opposée. Elle part de la phrase à analyser et cherche à remonter à l'axiome en réduisant une séquence de symboles terminaux et non-terminaux dans la protophrase courante vers le symbole non-terminal en partie gauche d'une règle de production ; ce remplacement produit la protophrase précédante dans la dérivation de la phrase.

L'analyse ascendante va chercher à produire une dérivation à droite. L'intérêt de ce choix est fort simple. Comme on construit la dérivation en remontant de la phrase vers l'axiome en considérant la phrase de gauche à droite, la règle de production que l'on cherche à chaque étape est celle qui apparie les symboles les plus à gauche possible dans la protophrase courante. Vu dans le sens normal de la dérivation, cela correspond donc au remplacement du dernier symbole non-terminal le plus à droite dans la protophrase courante, c'est-à-dire le plus à droite lorsque tout ceux qui étaient encore plus à droite ont déjà été réduits lors des dérivations précédentes. On constate donc que cette façon de procéder va naturellement construire une dérivation à droite, puisqu'en commençant par le symbole le plus à droite qui se trouve le plus à gauche dans la première protophrase lors de la construction d'une dérivation en sens inverse (vers l'axiome), on va aller peu à peu vers le plus à droite dans la première protophrase.

Pour construire une dérivation à droite en sens inverse, on va balayer les symboles de la protophrase courante de gauche à droite de manière à reconnaître le plus vite possible une partie droite de règle de production que l'on va pouvoir remplacer par sa partie gauche pour obtenir la protophrase précédente dans la dérivation. Tout le problème réside donc dans la reconnaissance de la partie droite de la bonne règle de production.

Exemple 6.5 Considérons la grammaire :

$$\begin{array}{rcl} S & \rightarrow & \texttt{a}AB\texttt{e} \\ A & \rightarrow & A\texttt{bc} \mid \texttt{b} \\ B & \rightarrow & \texttt{d} \end{array}$$

Illustrons l'analyse ascendante de la chaîne $\omega = \texttt{abbcde}$. On balaie la phrase de gauche à droite pour reconnaître que le b en seconde position peut être remplacé par A selon la seconde règle de production de ce non-terminal. On obtient alors la protophrase précédente aAbcde. On recommence alors pour se rendre compte que la séquence Abc correspond à la partie droite de la première règle de production sur A. On remplace donc cette séquence pour obtenir la protophrase précédente aAde. Maintenant, c'est le d en troisième position qui peut être remplacé par B pour obtenir aABe. Nous avons alors la partie droite de la règle de production de l'axiome S, ce qui nous permet d'obtenir S. En résumé, on a :

$$\texttt{a}\underline{\texttt{b}}\texttt{bcde} \underset{d}{\Leftarrow} \texttt{a}\underline{A\texttt{bc}}\texttt{de} \underset{d}{\Leftarrow} \texttt{a}A\underline{\texttt{d}}\texttt{e} \underset{d}{\Leftarrow} \underline{\texttt{a}AB\texttt{e}} \underset{d}{\Leftarrow} S$$ □

Nous allons voir que pour une classe de grammaires non-contextuelles incluant strictement les grammaires LL, il est possible de faire cette reconnaissance en balayant les symboles de la partie droite et en se donnant un nombre borné k de symboles de prédictions dans la suite de la protophrase courante après le manche. C'est ainsi que cette approche d'analyse ascendante est appelée $LR(k)$ pour «*Left-to-right scanning of the input, producing a Rightmost derivation in reverse with k symbols of prediction*».

6.3.1 Algorithme d'analyse général

Supposons une instant que le problème de reconnaissance de la partie droite de la règle de production à appliquer est résolu. L'algorithme général d'analyse ascendante va consister à balayer de la protophrase courante de gauche à droite et à empiler ces symboles tant que l'on n'a pas balayer toute la partie droite de la règle de production à appliquer. Lorsqu'on a reconnu une partie droite de règle de production, on va l'appliquer en dépilant ses symboles pour empiler à la place le non-terminal en partie gauche de ladite règle. Si on poursuit ce processus jusqu'à ce que l'axiome soit empilé et que la phrase ait été entièrement balayée, alors la phrase sera acceptée. Si on n'arrive pas à reconnaître une partie droite de règle de production ou encore si on termine les symboles de la protophrase courante en laissant une pile qui ne contient pas que l'axiome, alors la phrase est rejetée.

Revenons maintenant au problème de la reconnaissance de la partie droite de la bonne règle de production dans la protophrase courante. En terme de vocabulaire, on parle de *manche* pour désigner cette partie droite :

> **Manche :** sous-chaîne d'une protophrase droite de la grammaire qui correspond à la partie d'une règle de production dont l'application représente une étape le long

d'une dérivation droite inverse produite à l'aide de cette grammaire.

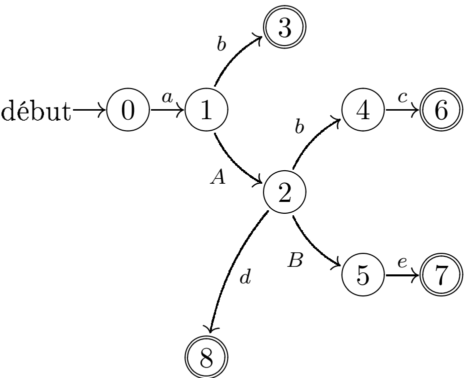
Un manche est donc une séquence de symboles terminaux et non-terminaux ainsi que sa position dans la protophrase droite. Notre problème de reconnaissance se résume donc au problème de reconnaissance des manches dans les protophrases droites productibles à partir de la grammaire. On parle d'*élagage* du manche lorsqu'on le remplace par la partie gauche de la règle de production correspondante.

L'idée (assez géniale) des analyses ascendantes du type LR est d'avoir reconnu que les manches d'une grammaire forment en réalité un langage régulier, et peuvent donc être reconnus par un automate d'états fini. Nous allons voir dans les sections suivantes comment construire l'automate qui reconnaît ces manches. Mais sachant que les manches sont reconnus par un automate, l'algorithme d'analyse va, pour chaque étape de dérivation inverse, consommer (et empiler) les symboles de la protophrase courante à l'aide de cet automate, et élaguer le manche dès que ce dernier est reconnu.

Exemple 6.6 Considérons à nouveau la grammaire précédente :

$$\begin{array}{lcl} S & \rightarrow & \texttt{a}AB\texttt{e} \\ A & \rightarrow & A\texttt{bc} \mid \texttt{b} \\ B & \rightarrow & \texttt{d} \end{array}$$

L'automate suivant reconnaît les manches dans le protophrases droites engendrées par cette grammaire :



Si on soumet à cet automate la protophrase droite `abbcde`, l'automate va de l'état 0 à l'état 1 sur `a`, puis vers l'état 3 sur `b`, qui est un état terminal signalant le manche `b`. De même, si on soumet la protophrase `a`A`bcde`, l'automate va de l'état 0 à 1 sur `a`, puis vers l'état 3 sur A, puis vers l'état 4 sur `b` et arrive à l"état final 6 sur `c`, ce qui signale le manche A`bc`. □

Bien sûr, il n'est pas nécessaire de reprendre le balayage de la protophrase courante depuis le début après chaque élagage. Un élagage peut être vu comme le fait de revenir en arrière dans l'automate jusqu'à l'état atteint avant de consommer les symboles du manche, puis à effectuer la transition sur le non-terminal en partie gauche de la règle de production appliquée. Pour faire ce chemin en sens inverse, on utilise une idée simple : on empile les états au fur et à mesure des transitions en les intercalant avec les symboles dans la pile. Ainsi, élaguer un manche va consister à dépiler autant d'états que de symboles dans le manche. Ceci va découvrir

en dessus de pile l'état atteint avant de consomer le manche, état duquel la transition sur le non-terminal empilé sera fait.

Pour mettre en œuvre cet algorithme général, on va se fonder sur deux tables de décision. Une table *Actions* indique, étant donné l'état en sommet de pile et le prochain symbole dans la protophrase courante s'il faut empiler ce dernier ou encore élaguer un manche qui se trouve en sommet de pile. Une table *Successeurs* va indiquer pour l'état en sommet de pile et un non-terminal à empiler, vers quel état la transition doit se faire dans l'automate.

Plus précisément, soit G une grammaire de laquelle des tables d'analyse *Actions* et *Successeurs* ont été construites et ω une chaîne à analyser, l'algorithme suivant va produire une dérivation à droite de w :

1. Positionner le curseur sur le premier symbole de $\omega\$$.
2. Initialiser la *pile* avec le symbole de l'état initial s_0 en fond de pile.
3. **répéter**
4.   soit s le symbole d'état en sommet de pile et a le prochain symbole dans $\omega\$$.
5.   **si** *Actions*$[s, a]$ est (d, s') **alors**
6.   Empiler a, puis s' et avancer le curseur dans $\omega\$$
       sinon
7   **si** *Actions*$[s, a]$ est $(r, A \to \beta)$ **alors**
8.   Dépiler $2 \times |\beta|$ symboles de la pile.
9.   Soit s' le symbole d'état maintenant en sommet de pile.
10.   Empiler A sur la pile.
11.   Empiler le symbole d'état *Successeur*$[s', A]$.
            sinon
12.   **si** *Actions*$[s, a]$ est *accepter* **alors**
13.   accepté := **vrai**
                sinon
14.   Déclencher une erreur de syntaxe.
15. **jusqu'à** accepté.

Exemple 6.7 Considérons la grammaire des expressions arithmétiques suivante :

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathtt{id} \end{array}$$

Une dérvation de la phrase $\mathtt{id}_1\mathtt{+id}_2\mathtt{*id}_3$ donnerait lieu aux actions suivantes :

Pile	Entrée	Action
$	id_1+id_2*$id_3$$	
$ id_1	+id_2*$id_3$$	décaler id_1
$ F	+id_2*$id_3$$	manche, élaguer
$ T	+id_2*$id_3$$	manche, élaguer
$ E	+id_2*$id_3$$	manche, élaguer
$ E+	id_2*$id_3$$	décaler +
$ E+id_2	*$id_3$$	décaler id_2
$ E+F	*$id_3$$	manche, élaguer
$ E+T	*$id_3$$	manche, élaguer
$ E+T*	$id_3$$	décaler *
$ E+T*id_3	$	décaler id_3
$ E+T*F	$	manche, élaguer
$ E+T	$	manche, élaguer
$ E	$	manche, élaguer
$ E	$	accepter

□

Toute la difficulté des méthodes ascendantes, qui réside dans la prise des décisions d'analyse (décaler, élaguer, accepter), revient à la détection du manche. Les méthodes LR sont fondées sur l'utilisation de tables d'analyse qui sont construites à partir des grammaires. Trois méthodes sont maintenant proposées, qui diffèrent essentiellement par leur capacité à traiter des classes plus ou moins importantes de grammaires non-contextuelles.

6.3.2 Construction des tables d'analyse ascendante de type LR

Construction des tables SLR

à compléter

Construction des tables LR canoniques

Construction des tables LALR

Soit G une grammaire, l'algorithme produit des tables d'analyse SLR *Actions* et *Successeurs*, si la grammaire est SLR. Pour cela, il faut calculer la collection des ensembles d'items SLR pour G, puis la fonction SUIVANT pour chacun des symboles non-terminaux de G, et enfin construire les tables elles-mêmes à partir de cette collection et de cette fonction.

Collection des ensembles d'items SLR

Soit $A \rightarrow XYZ$ une règle de production de la grammaire G, un item SLR est une production avec un point repérant une position dans sa partie droite. Le point est utilisé pour indiquer la portion de partie droite déjà reconnue dans une protophrase droite (i.e. la séquence de symboles déjà reconnus).

Opération $Fermeture(I)$

Soit I un ensemble d'items SLR, la fermeture de I, notée $Fermeture(I)$, est obtenue par les deux règles suivantes :

1. Initialement, placer chaque item de I dans $Fermeture(I)$.
2. Si $[A \rightarrow \alpha \bullet B\beta]$ est dans $Fermeture(I)$ et $B \rightarrow \gamma$ est une production de G, alors ajouter l'item $[B \rightarrow \bullet\gamma]$ à $Fermeture(I)$ s'il ne s'y trouve pas déjà. Répéter l'application de cette deuxième règle jusqu'à ce qu'il ne soit plus possible d'ajouter d'items à $Fermeture(I)$.

Opération $Transition[I, X]$

Soit I un ensemble d'items et X un symbole de G,

1. Calculer I', l'ensemble des items $[A \rightarrow \alpha X \bullet \beta]$ tels que l'item $[A \rightarrow \alpha \bullet X\beta] \in I$.
2. $Transition[I, X] = Fermeture(I')$.

Collection des ensembles d'items LR(0) pour G

Soit $G' = (V', T, P', S')$ la grammaire augmentée obtenue de $G = (V, T, P, S)$ en ajoutant le non-terminal S' à V pour obtenir V', la production $S' \rightarrow S$ à P pour obtenir P' et en remplaçant l'axiome S par S'.

1. Initialiser C au seul ensemble d'items non-marqué obtenu par $Fermeture(\{[S' \rightarrow \bullet S]\})$.
2. **tant que** il existe des ensemble d'items non-marqués dans C **faire**
3.   Soit I un ensemble d'items non-marqué de C.
4.   **pour** chaque symbole de grammaire $X \in V \cup T$ **faire**
5.   **si** il existe un item $[A \rightarrow \alpha \bullet X\beta]$ dans I **alors**
6.   S'il n'y est pas déjà, ajouter $Transition[I, X]$ comme ensemble non-marqué à C.

    fin.

fin.

FIG. 6.7 – Construction des tables d'analyse SLR.

Construction des tables d'analyse SLR *Actions* **et** *Successeurs*
Soit G' une grammaire augmentée, les tables d'analyse SLR *Actions* et *Successeurs* sont obtenues par :

1. Construire $C = \{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(0) pour la grammaire G'.
2. L'état i est construit à partir de l'ensemble d'items I_i. Les actions d'analyse pour l'état i sont déterminées par les règles suivantes :
   (a) Si $[A \rightarrow \alpha \bullet a\beta] \in I_i$, si $a \in T$ et si $Transition[I_i, a] = I_j$, alors $Action[i, a] =$ *décaler* j.
   (b) Si $[A \rightarrow \alpha\bullet] \in I_i$, $\forall a \in Suivants(A)$, $Action[i, a] =$ *réduire par* $A \rightarrow \alpha$.
   (c) Si $[S' \rightarrow S\bullet] \in I_i$, alors $Action[i, \$] =$ *accepter*.

   Si les règles précédentes sont conflictuelles, alors la grammaire n'est pas SLR et l'algorithme échoue.
3. Si $A \in V$ et $Transition[I_i, A] = I_j$ alors $Successeurs[i, A] = j$.
4. Toutes les entrées non-définies par les règles en (2) et (3) sont positionnées à *erreur*.
5. L'état initial de l'analyseur est celui qui a été construit à partir de l'ensemble contenant l'item $[S' \rightarrow \bullet S]$.

FIG. 6.8 – Construction des tables d'analyse SLR (suite).

Soit G une grammaire, l'algorithme produit des tables d'analyse LR *Actions* et *Successeurs*, si la grammaire est LR(1). Pour cela, il faut calculer la collection des ensembles d'items LR(1) pour G, puis construire les tables elles-mêmes à partir de cette collection.

Collection des ensembles d'items LR(1)

Soit $A \rightarrow XYZ$ une règle de production de la grammaire G, un item LR(1) est un item LR(0) auquel est ajouté un ensemble de symboles terminaux $\{a_1, ..., a_n\}$ pouvant suivre le non-terminal A dans une protophrase droite obtenue dans le contexte où A a été remplacé par la partie droite XYZ de la règle de production. Par exemple, l'item $[A \rightarrow X \bullet YZ, a_1/.../a_n]$ indique que le non-terminal A a été remplacé par XYZ dans une protophrase droite dans laquelle A pouvait être suivie par un des symboles terminaux $a_1, ..., a_n$, et que dans cette partie droite, l'algorithme a reconnu une séquence de symboles correspondant au non-terminal X.

Opération $Fermeture(I)$

Soit I un ensemble d'items LR, la fermeture de I, notée $Fermeture(I)$, est obtenue par les deux règles suivantes :

1. Initialement, placer chaque item de I dans $Fermeture(I)$.
2. Si $[A \rightarrow \alpha \bullet B\beta, a]$ est dans $Fermeture(I)$ et $B \rightarrow \gamma$ est une production de G, alors ajouter l'item $[B \rightarrow \bullet\gamma, b]$ à $Fermeture(I)$ pour chaque terminal b dans PREMIER(βa), s'il ne s'y trouve pas déjà. Répéter l'application de cette deuxième règle jusqu'à ce qu'il ne soit plus possible d'ajouter d'items à $Fermeture(I)$.

Opération $Transition[I, X]$

Soit I un ensemble d'items et X un symbole de G,

1. Calculer I', l'ensemble des items $[A \rightarrow \alpha X \bullet \beta, a]$ tels que $[A \rightarrow \alpha \bullet X\beta, a] \in I$.
2. $Transition[I, X] = Fermeture(I')$.

Collection des ensembles d'items LR(1) pour G

Soit $G' = (V', T, P', S')$ la grammaire augmentée obtenue de $G = (V, T, P, S)$ en ajoutant le non-terminal S' à V pour obtenir V', la production $S' \rightarrow S$ à P pour obtenir P' et en remplaçant l'axiome S par S'.

1. Initialiser C au seul ensemble d'items non-marqué obtenu par $Fermeture(\{[S' \rightarrow \bullet S, \$]\})$.
2. **tant que** il existe des ensemble d'items non-marqués dans C **faire**
3. Soit I un ensemble d'items non-marqué de C.
4. **pour** chaque symbole de grammaire $X \in V \cup T$ **faire**
5. **si** il existe un item $[A \rightarrow \alpha \bullet X\beta, a]$ dans I **alors**
6. S'il n'y est pas déjà, ajouter $Transition[I, X]$ comme ensemble non-marqué à C.

fin.

fin.

FIG. 6.9 – Construction des tables LR canoniques.

Construction des tables d'analyse LR canoniques $Actions$ **et** $Successeurs$
Soit G' une grammaire augmentée, les tables d'analyse LR canoniques $Actions$ et $Successeurs$ sont obtenues par :

1. Construire $C = \{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(1) pour la grammaire G'.
2. L'état i est construit à partir de l'ensemble d'items I_i. Les actions d'analyse pour l'état i sont déterminées par les règles suivantes :
   (a) Si $[A \to \alpha \bullet a\beta, b] \in I_i$ avec $a \in T$ et si $Transition[I_i, a] = I_j$, alors $Action[i, a] = \textit{décaler}\, j$.
   (b) Si $[A \to \alpha\bullet, a] \in I_i$, $A \neq S'$ $Action[i, a] = \textit{réduire par}\ A \to \alpha$.
   (c) Si $[S' \to S\bullet, \$] \in I_i$, alors $Action[i, \$] = accepter$.

   Si les règles précédentes sont conflictuelles, alors la grammaire n'est pas LR(1) et l'algorithme échoue.
3. Si $A \in V$ et $Transition[I_i, A] = I_j$ alors $Successeurs[i, A] = j$.
4. Toutes les entrées non-définies par les règles en (2) et (3) sont positionnées à *erreur*.
5. L'état initial de l'analyseur est celui qui a été construit à partir de l'ensemble contenant l'item $[S' \to \bullet S, \$]$.

FIG. 6.10 – Construction des tables LR canoniques (suite).

Construction des tables d'analyse LALR $Actions$ **et** $Successeurs$
Un algorithme facile, mais gourmand en place consiste à construire d'abord les items LR(1), puis à fusionner les ensembles d'items de même noyau (ensembles d'items identiques, aux symboles de prévision près).

1. Construire $C = \{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(1) pour la grammaire G'.
2. Pour chaque noyau présent parmi les ensembles d'items LR(1), trouver tous les états ayant ce même noyau et remplacer ces états par leur union.
3. Soit $C' = \{J_0, J_1, ..., J_n\}$, la collection des ensembles d'items LR(1) résultante. Pour chacun de ceux-ci, déterminer les actions d'analyse comme dans l'algorithme pour construire les tables LR canoniques. Si ces règles mènent à un conflit, la grammaire n'est pas LALR(1) et l'algorithme échoue.
4. La fonction $Transition$ est construite comme suit. Soit $J = I_0 \cup I_1 \cup ... \cup I_k$, les noyaux de $Transition[I_0, X]$, $Transition[I_1, X]$, ... et $Transition[I_k, X]$ sont les mêmes puisque $I_0, I_1, ..., I_k$ ont le même noyau. Soit K l'union de tous les ensembles d'items ayant le même noyau que $Transition[I_0, X]$. Alors $Transition[J, X] = K$.
5. La table $Successeur$ est construite comme pour celle des tables LR canoniques, mais sur la collection d'ensembles d'items C'.

FIG. 6.11 – Construction des tables LALR.

Chapitre 7

Analyse sémantique, grammaires attribuées et transformations

Le résultat de l'analyse d'une phrase est une dérivation qui peut alternativement être représentée par un arbre de dérivation. L'arbre de dérivation explicite la structure de la phrase en terme de remplacements successifs des non-terminaux par des séquences de terminaux et non-terminaux, à la façon de l'analyse des langages naturels (français, par exemple) où une phrase se compose d'un sujet, d'un verbe et d'un complément, etc. L'analyse montre que la phrase fait bien partie du langage engendré par la grammaire. À partir de là, on peut s'intéresser à d'autres propriétés grammaticales. Dans le domaine des langages de programmation, on peut s'intéresser à la concordance des types par exemple. Ce genre de vérifications de propriétés grammaticales sont réalisées par ce que l'on appelle des *analyses sémantiques* :

> **Analyse sémantique :** calcul de propriétés sur un arbre de dérivation

De même, on peut transformer la phrase en une autre phrase, éventuellement dans un autre langage, par traduction ou compilation. Les analyses sémantiques et les traductions partagent une approche commune qui consiste à être définies par induction sur la structure de la phrase, donc sur la forme de l'arbre de dérivation. Un des cadres théoriques qui a été développé pour définir des analyse ou des transformations par induction sur la structure des phrases s'appelle les *grammaires attribuées* :

> **Grammaire attribuée :** grammaire non-contextuelle pour laquelle on associe un ou des attributs aux symboles et où à chaque règle de production on associe une ou plusieurs règles de calcul permettant de calculer la valeur des attributs des symboles y apparaissant.

Les grammaires attribuées sont largement utilisées en compilation des langages de programmation, mais plus généralement elles ont de nombreuses applications pratiques dans l'industrie. Dans ce chapitre, nous allons voir deux réalisations de grammaires attribuées : les définitions dirigées par la syntaxe et les schémas de traduction.

7.1 Définitions dirigées par la syntaxe

Il existe deux grands types d'attributs dans les grammaires attribuées : les *attributs synthétisés* et les *attributs hérités*.

Attribut synthétisé : attribut d'un symbole non-terminal en partie gauche de règle de production dont la valeur est calculée à partir des valeurs des attributs des symboles apparaissant en partie droite de ses règles de production.

Attribut hérité : attribut d'un symbole non-terminal en partie droite d'une règle de production dont la valeur est calculée à partir des valeurs des attributs des autres symboles de la règle de production (y compris le symbole en partie gauche).

Plus explicitement, pour une production $A \to X_1X_2...X_n$, une règle sémantique $b = f(c_1, ..., c_k)$ peut calculer un attribut synthétisé b du non-terminal A à partir des attributs $c_1, ..., c_k$ des symboles $X_1, ..., X_n$, ou encore calculer la valeur d'un attribut hérité b d'un symbole X_i à partir des attributs $c_1, ...c_k$ des symboles $A, X_1, ...X_n$.

Exemple 7.1 Soit la grammaire :

$$\begin{array}{lcl} L & \to & E\texttt{=} \\ E & \to & E\texttt{+}T \mid T \\ T & \to & T\texttt{*}F \mid F \\ F & \to & \texttt{(}E\texttt{)} \mid n \end{array}$$

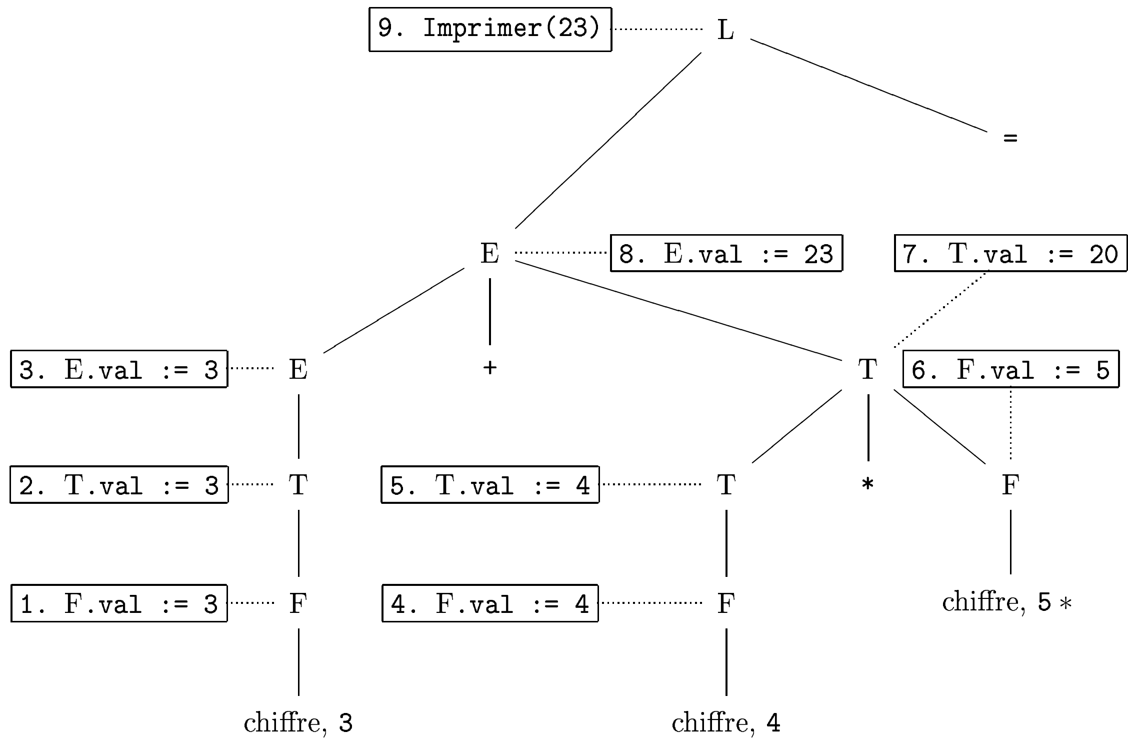
où n est un nombre entier. On peut écrire une définition dirigée par la syntaxe pour calculer la valeur d'un attribut `val` représentant la valeur de l'expression de la manière suivante :

Production	Règle sémantique
$F \to n$	F`.val := n.vallex`
$F \to$ `(`E`)`	F`.val :=` E`.val`
$T \to F$	T`.val :=` F`.val`
$T \to T_1$`*`F	T`.val :=` T_1`.val`$\times F$`.val`
$E \to T$	E`.val :=` T`.val`
$E \to E_1$`+`T	E`.val :=` E_1`.val`$+T$`.val`
$L \to E$`=`	`Imprimer(`E`.val)`

où l'attribut `vallex` est un attribut calculé lors de la reconnaissance du terminal n (typiquement lors de l'analyse lexicographique dans un langage de programmation). □

Les attributs synthétisés possèdent une propriété très importante en ce qui concerne leur ordre d'évaluation dans un arbre de dérivation. En effet, comme l'attribut de chaque non-terminal peut être calculé à partir des valeurs d'attributs de ses fils, il est possible d'évaluer tous les attributs synthétisés en parcourant l'arbre depuis les feuilles jusqu'à la racine. Cet ordre est exactement celui que suit l'analyse ascendante de type LR. Il est donc possible d'évaluer ces attributs à la volée, c'est-à-dire pendant la création de l'arbre de dérivation lors de l'analyse.

Exemple 7.2 Pour la grammaire attribuée précédente, calculons les attributs sur la phrase `3+4*5=` :



Les numéros en étiquettes indiquent l'ordre par lequel les attributs sont calculés. □

Les attributs synthétisés sont les plus simples à comprendre et à calculer. Il sont beaucoup utilisés dans toutes les définitions de transformations qui peuvent s'apparenter à de l'évaluation au sens large. Les attributs hérités, par contre, ont une grande importance pour modéliser toute propriété où il y a dépendance envers le contexte. C'est le cas par exemple de l'obligation à déclarer une variable avant de l'utiliser ou le typage des expressions dans les langages de programmation.

Exemple 7.3 Cet exemple illustre l'utilisation d'attributs hérités pour traiter des règles de typage dans un langage de programmation. Soit la grammaire :

$$\begin{array}{lcl} D & \rightarrow & T\ L \\ T & \rightarrow & \texttt{entier} \mid \texttt{réel} \\ L & \rightarrow & L\texttt{,id} \mid \texttt{id} \end{array}$$

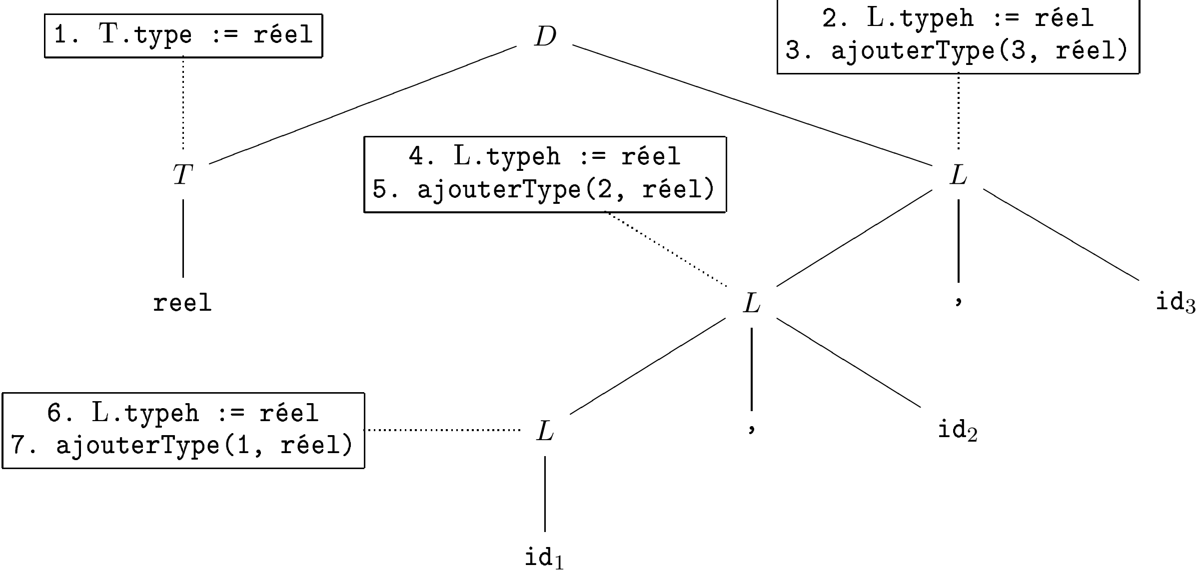
où une déclaration D est formée d'un nom de type T et d'une liste d'identificateurs de variables L. La grammaire attribuée suivante évalue des attributs `type` et `typeh` (pour type hérité) :

Production	Règle sémantique
$D \rightarrow T\ L$	L`.typeh := `T`.type`
$T \rightarrow$ `entier`	T`.type := entier`
$T \rightarrow$ `réel`	T`.type := réel`
$L \rightarrow L_1$`,id`	L_1`.typeh := `L`.typeh`
	et `ajouterType(id.entree, `L`.typeh)`
$L \rightarrow$ `id`	`ajouterType(id.entree, `L`.typeh)`

où la procédure `ajouterType` va renseigner le champ type associée à l'entrée de la table des symboles correspondant à la variable. Dans un compilateur, la table des symboles est une

structure de données primordiale qui centralise toutes les informations relatives aux identificateurs qui sont nécessaires pour la traduction vers le langage machine.

Considérons maintenant l'évaluation des attributs `type` et `typeh` pour la phrase '`réel` $\texttt{id}_1$`,` $\texttt{id}_2$`,` $\texttt{id}_3$'.



Encore cette fois, les numéros en étiquettes indiquent l'ordre selon lequel les attributs sont calculés. □

Méthode générale d'évaluation des attributs

Nous avons vu que l'ordre d'évaluation le plus simple pour les attributs est celui des attributs synthétisés. Il suffit d'évaluer les attributs en partant des feuilles de l'arbre de dérivation en remontant vers la racine. Nous avons également fait remarquer que cet ordre d'évaluation correspond parfaitement à l'ordre de création des nœuds de l'arbre de dérivation dans les analyses ascendantes de type LR.

> **Grammaire S-attribuée :** une grammaire S-attribuée est une grammaire attribuée dont tous les attributs sont synthétisés.

Grammaires S-attribuées et analyse ascendante LR forment un couple d'outils idéal en terme d'efficacité, puisque l'évaluation des attributs se fait à la volée au cours de la génération de l'arbre. En fait, il n'est même pas nécessaire de créer l'arbre de dérivation pour calculer les attributs synthétisés.

Dans le cas général, l'ordre d'évaluation nécessaire pour les attributs peut être totalement différent de celui de la création des nœuds de l'arbre de dérivation. En fait, chaque règle sémantique établit des dépendances entre les valeurs d'attributs. La méthode générale pour l'évaluation des attributs d'une grammaire attribuée consiste donc à créer un graphe où les nœuds sont les valeurs d'attributs à calculer et les arcs représentent les relations de dépendances entre ces valeurs. Si le graphe obtenu est sans cycle, on peut déterminer l'ordre d'évaluation en appliquant un algorithme dit de *tri topologique*, c'est-à-dire un algorithme qui produit une séquence «croissante» de nœuds suivant l'ordre de dépendance.

Si le graphe de dépendance contient un cycle, alors le calcul des attributs fait apparaître une dépendance circulaire d'une valeur d'attribut envers elle-même. Dans certains de ces cas, il est possible d'appliquer une technique dite de *point fixe* où on donne une valeur initiale pour un certain nœud dans le graphe de dépendance, puis on calcule répétitivement les attributs jusqu'à ce que les valeurs obtenues ne changent plus ; on a alors atteint le point fixe. Rares sont les applications pratiques où on utilise ces techniques, car elles sont généralement très coûteuses en temps et en espace mémoire.

7.2 Schéma de traduction

Entre les grammaires S-attribuées et les grammaires attribuées où l'ordre d'évaluation est quelconque, on trouve d'autres cas où l'ordre est relativement facile à obtenir et à mettre en œuvre. En particulier, plusieurs grammaires attribuées avec attributs hérités présentent la propriété que tous les attributs hérités ne dépendent que des attributs des symboles à leur gauche dans la règle de production (y compris le symbole en partie gauche de la règle. Dans ce cas, l'ordre d'évaluation des attributs peut être rendu compatible avec un ordre ascendant de création des nœuds.

> **Grammaire L-attribuée :** une grammaire attribuée est dite L-attribuée si tout attribut hérité de X_j, $1 \leq j \leq n$, de la partie droite d'une règle de production $A \rightarrow X_1, ..., X_n$ ne dépend que :
>
> 1. des attributs des symboles $X_1, ...X_{j-1}$ situés à gauche de X_j, et
> 2. des attributs hérités de A.

Parce que les règles sémantiques des grammaires L-attribuées font apparaître des dépendances entre attributs des non-terminaux d'une même règle ainsi qu'entre symboles d'une règle et ceux du symbole en partie gauche de la même règle, il est important d'évaluer les règles au fur et à mesure de la traversée de l'arbre de dérivation. On ne peut en effet attendre d'avoir parcouru toute la partie droite de la règle pour évaluer les attributs car les attributs hérités ne seraient pas disponibles pour kes réductions intermédiaires. Pour expliciter cet ordre, on utilise un autre formalisme de définition des grammaires L-attribuées que l'on appelle les *schémas de traduction* :

> **Schéma de traduction :** Un schéma de traduction est une grammaire non-contextuelle où des actions sémantiques, délimitées par des accolades, sont insérées dans les parties droites des règles de production.

Exemple 7.4 Considérons la grammaire des expressions arithmétiques que nous avons introduite au début de ce chapitre :

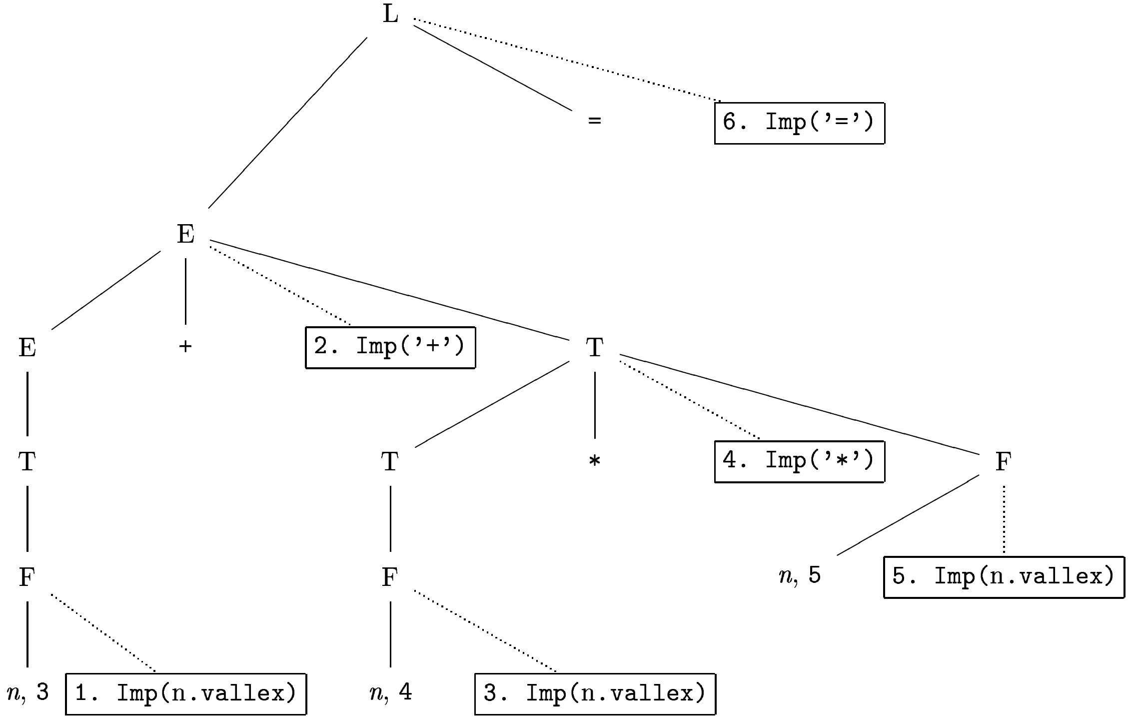
$$
\begin{array}{lcl}
L & \rightarrow & E\texttt{=} \\
E & \rightarrow & E\texttt{+}T \mid T \\
T & \rightarrow & T\texttt{*}F \mid F \\
F & \rightarrow & \texttt{(}E\texttt{)} \mid n
\end{array}
$$

Un schéma de traduction qui réalise la resynthèse d'un source de programme est défini de la

manière suivante :

$$
\begin{array}{lcl}
L & \rightarrow & E\texttt{=}\ \{\texttt{Imp('=')}\} \\
E & \rightarrow & E\texttt{+}\ \{\texttt{Imp('+')}\}\ T \mid T \\
T & \rightarrow & T\texttt{*}\ \{\texttt{Imp('*')}\}\ F \mid F \\
F & \rightarrow & \texttt{(}\ \{\texttt{Imp('(')}\}\ E\texttt{)}\ \{\texttt{Imp(')')}\} \mid n\ \{\texttt{Imp(}n\texttt{.vallex)}\}
\end{array}
$$

où `Imp` signifie imprimer. Considérons la phrase `3+4*5=` dont l'arbre de dérivation décoré des actions sémantiques est :



L'ordre dans lequel les règles sémantiques sont évaluées est donné par le numéro qui leur est associé. Cela correspond à une visite de gauche à droite en profondeur d'abord de l'arbre de dérivation. □

Le principal avantage des grammaires L-attribuées tient à ce que l'ordre d'évaluation des attributs suit justement ce parcours de gauche à droite en profondeur d'abord de l'arbre de dérivation. Il s'agit d'un ordre simple à programmer. De plus, nous avons vu comment l'évaluation des attributs synthétisés peut se faire en remontant l'arbre depuis les feuilles jusqu'à la racine et comment cet ordre se marie parfaitement à l'ordre de création de nœuds dans une analyse ascendante de type LR. Il se trouve qu'on peut faire coller l'ordre d'évaluation d'une grammaire L-attribuée à cet ordre de création ascendante des nœuds.

Pour arriver à cela il se pose deux problèmes : déclencher l'évaluation des règles sémantiques lors de la réduction d'une partie droite vers une partie gauche de règle de production et récupérer les valeurs des attributs hérités calculés pour les symboles plus à gauche dans la règle de production (ou venant de la partie gauche de la règle). Voyons ces deux problèmes tour à tour.

7.2.1 Élimination des actions intérieures en faveur d'actions à la réduction

Dans une analyse ascendante, toutes les règles sémantiques sont évaluées lors de la réduction d'une partie droite de règle de production vers sa partie gauche. Dans une grammaire L-attribuée (ou un schéma de traduction), des règles doivent être évaluées entre la reconnaissance de deux symboles successifs dans la partie droite des règles de production. Pour obtenir cet effet au moment d'une réduction, il suffit d'introduire pour chaque règle intérieure à évaluer un non-terminal bidon, dérivant la phrase vide, dont le rôle sera uniquement de déclencher l'évaluation de la règle sémantique qui lui est associée.

Exemple 7.5 Reprenons l'exemple précédent de la synthèse du source. Pour chaque règle sémantique qui n'apparaît pas à la fin de la partie droite de la règle de production, on introduit un non-terminal bidon, dérivant ϵ auquel est associé la règle intérieure :

$$
\begin{array}{lcl}
L & \rightarrow & E\texttt{=}\ \{\texttt{Imp('=')}\} \\
E & \rightarrow & E\texttt{+}X\ T \ \mid\ T \\
X & \rightarrow & \epsilon\ \{\texttt{Imp('+')}\} \\
T & \rightarrow & T\texttt{*}Y\ F \ \mid\ F \\
Y & \rightarrow & \epsilon\ \{\texttt{Imp('*')}\} \\
F & \rightarrow & \texttt{(}Z\ E\texttt{)}\ \{\texttt{Imp(')')}\} \ \mid\ n\ \{\texttt{Imp(}n\texttt{.vallex)}\} \\
Z & \rightarrow & \epsilon\ \{\texttt{Imp('(')}\}
\end{array}
$$

Toutes les actions sont alors à la fin des parties droites de règles de production. On peut donc les évaluer à la réduction, comme pour les attributs synthétisés. On remarque aussi que l'ordre d'évaluation des règles pour une analyse de la phrase `3+4*5=` est équivalente à celle du schéma de traduction précédent. □

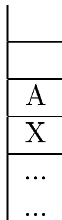
7.2.2 Héritage des attributs dans la pile d'analyse

La transformation précédente est très simple à réaliser. Elle fonctionne sans difficulté si les règles sémantiques se bornent à faire des effets de bord, comme dans le cas précédent. Cependant, s'il s'agit effectivement d'évaluer des attributs hérités, elle est insuffisante. Pour comprendre le problème, considérons deux règles de production avec des règles sémantiques associées :

$$
\begin{array}{ll}
A \rightarrow X\ Y & Y\texttt{.h := }X\texttt{.s} \\
Y \rightarrow A\ B & A\texttt{.h := }Y\texttt{.h}
\end{array}
$$

où `h` est un attribut hérité de Y et A alors que `s` est un attribut synthétisé de X. Le problème qui se pose est que l'attribut `h` de Y doit être évalué avant de commencer l'analyse de la portion de phrase qui va être réduite à Y car, par définition d'une grammaire L-attribuée, cet attribut peut être utilisé dans les règles sémantiques associées aux productions de Y. En particulier, on peut l'utiliser dans les règles évaluant les attributs de A ou de B.

L'idée ici est d'utiliser le fait que la valeur d'attribut Y`.h` est obtenue directement de l'attribut X`.s` par simple recopie pour récupérer directement cette valeur dans la pile de l'analyseur ascendant. En effet, en analyse ascendante, on reconnaît d'abord la portion de phrase correspondant au symbole X puis celle correspondant à la portion de phrase dérivable à partir du symbole Y. Comme il faut reconnaître ce qui correspondant au symbole A avant de réduire vers Y, on peut se retrouver dans la configuration de pile d'analyse suivante :

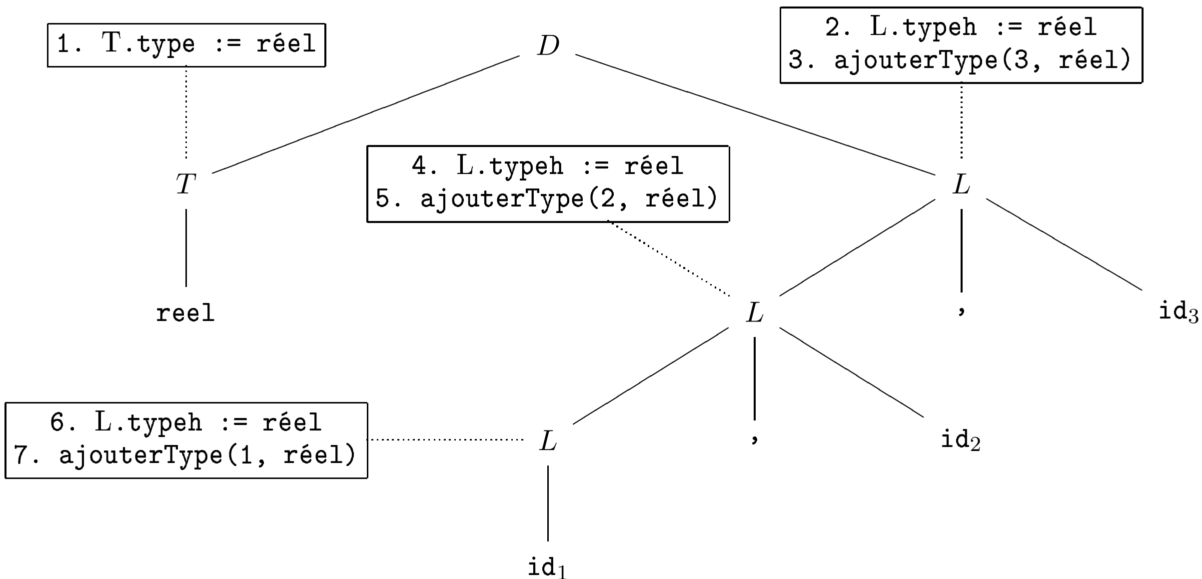


Supposons qu'à cet instant, il faille évaluer une règle nécessitant la valeur de Y.h. Le symbole Y n'est pas dans la pile mais X lui y est déjà. Il suffit donc de remplacer l'accès à Y.h par un accès à l'attribut s du symbole X présent lui dans la pile. Pour cela, on remarque que la position du symbole X dans la pile par rapport au sommet sera toujours la même lorsque la règle sémantique doit être évaluée : à la position sommet - 1. On remplace donc dans la règle sémantique l'accès Y.h par `pile[sommet-1].s`, en supposant que l'on peut avoir ainsi accès au contenu de la pile comme s'il s'agissait d'un tableau.

Exemple 7.6 Reprenons la grammaire des déclarations et sa définition dirigée par la syntaxe pour la propagation du type des variables :

Production	Règle sémantique
$D \rightarrow T\ L$	L.typeh := T.type
$T \rightarrow$ entier	T.type := entier
$T \rightarrow$ réel	T.type := réel
$L \rightarrow L_1$,id	L_1.typeh := L.typeh
	et ajouterType(id.entree, L.typeh)
$L \rightarrow$ id	ajouterType(id.entree, L.typeh)

Sur l'exemple 'réel p, q, r', nous avions obtenu l'arbre de dérivation suivant :



Voyons comment se passe l'analyse ascendante et comment l'évaluation des règles sémantiques butent sur l'absence de l'attribut hérité dans la pile :

Pile	Phrase	Production	Règle sémantique
	réel p, q, r		
réel	p, q, r		
T(`type = réel`)	p, q, r	$T \rightarrow$ `réel`	T`.type := réel`
T(`type = réel`) p	, q, r		
T(`type = réel`) L	, q, r	$L \rightarrow$ `id`	`ajouterType(id.entree,` L`.typeh)`

À cette étape, on ne peut exécuter la règle sémantique car l'attribut L`.typeh` n'est pas disponible sur la pile. Par contre la valeur de l'attribut T`.type`, elle, est disponible au deuxième symbole en partant du sommet de la pile. Supposons que l'on puisse utiliser une fonction `accederPile` avec un argument entier donnant le symbole à accéder à partir du sommet de la pile, on peut réécrire la grammaire attribuée précédente de la façon suivante :

Production	Règle sémantique
$D \rightarrow T\ L$	
$T \rightarrow$ `entier`	T`.type := entier`
$T \rightarrow$ `réel`	T`.type := réel`
$L \rightarrow L_1$`,id`	`ajouterType(id.entree, accederPile(4).type)`
$L \rightarrow$ `id`	`ajouterType(id.entree, accederPile(2).type)`

Les règles de recopie étant devenue inutiles, on les a simplement éliminées. Voici comment se passe alors l'analyse ascendante :

Pile	Phrase	Production	Règle sémantique
	réel p, q, r		
réel	p, q, r		
T(`type = réel`)	p, q, r	$T \rightarrow$ `réel`	T`.type := réel`
T(`type = réel`) p	, q, r		
T(`type = réel`) L	, q, r	$L \rightarrow$ `id`	`ajouterType(id.entree,` `accederPile(2).type)`
T(`type = réel`) L ,	q, r		
T(`type = réel`) L , q	, r		
T(`type = réel`) L	, r	$L \rightarrow L_1$`,id`	`ajouterType(id.entree,` `accederPile(4).type)`
T(`type = réel`) L ,	r		
T(`type = réel`) L , r			
T(`type = réel`) L		$L \rightarrow L_1$`,id`	`ajouterType(id.entree,` `accederPile(4).type)`

On note que les règles sémantiques étant évaluées juste au moment de la réduction, le nombre passé en paramètre à la fonction `accederPile` correspond bien à chaque fois à la position du symbole T dans la pile. □

La technique d'accès dans la pile est bien adaptée à beaucoup de grammaires L-attribuées où les attributs hérités sont souvent obtenus par des règles de recopie d'attributs synthétisés. Par contre, cette technique impose que le symbole que l'on doit accéder dans la pile soit toujours à distance fixe par rapport au sommet de la pile. Ce n'est malheureusement pas toujours le cas. Considérons la grammaire L-attribuée suivante :

Production	Règle sémantique
$S \rightarrow \texttt{a}\ A\ C$	C.h := A.s
$S \rightarrow \texttt{b}\ A\ B\ C$	C.h := A.s
$C \rightarrow D$	D.h := g(C.h)

Le problème qui se pose ici est que lors de l'évaluation de la règle sémantique D.h := g(C.h), on cherche à remplacer C.h par un accès au symbole A dans la pile. Malheureusement, on peut se trouver dans deux cas de figure :

- soit l'analyse tente de reconnaître avec la première règle de production sur S, et alors le symbole A est le deuxième symbole dans la pile,
- soit l'analyse tente de reconnaître avec la deuxième règle de production sur S, et alors le symbole A est le troisième symbole dans la pile.

Comme on doit avoir une seule règle sémantique associée à une production, on ne peut pas remplacer C.h à la fois par `accederPile(2).s` et par `accederPile(3).s`. Dans ce cas, l'ajout d'un non-terminal marqueur peut solutionner le problème. Considérons la grammaire L-attribuée modifiée :

Production	Règle sémantique
$S \rightarrow \texttt{a}\ A\ C$	C.h := A.s
$S \rightarrow \texttt{b}\ A\ B\ M\ C$	M.h := A.s
	C.h := M.s
$M \rightarrow \epsilon$	M.s := M.h
$C \rightarrow D$	D.h := g(C.h)

Ici, il devient possible de remplacer M.h par A.s, c'est-à-dire `accederPile(2).s` car lors de la réduction de ϵ vers M, le symbole A se trouve immédiatement sous le symbole B qui est en sommet de pile. De même, on peut remplacer C.h par A.s et M.s selon le cas, c'est-à-dire à chaque fois par `accederPile(2).s` car dans les deux cas de figure, M ou A se trouvent en deuxième position à partir du sommet de pile lors de l'évaluation de la règle. En enlevant les règles de recopie devenues inutiles, on obtient alors la grammaire attribuée suivante :

Production	Règle sémantique
$S \rightarrow \texttt{a}\ A\ C$	
$S \rightarrow \texttt{b}\ A\ B\ M\ C$	
$M \rightarrow \epsilon$	M.s := accederPile(2).s
$C \rightarrow D$	D.h := g(accederPile(2).s)

Chapitre 8

Transformations de documents XML

La popularité d'XML tient à l'indépendance des documents de la façon dont ils seront présentés ou exploités. L'information est balisée sémantiquement, ce qui permet de la repérer facilement dans un document (pas besoin d'analyse lexicale ou syntaxique pour ce faire). La puissance de XML tient donc à la facilité avec laquelle on va pouvoir récupérer de l'information dans un document de manière à l'exploiter, pour la présenter ou pour produire de nouveaux documents XML. Il existe deux grandes approches pour traiter les documents XML : la transformation de document à partir d'un langage de haut niveau comme XSL et le traitement de document comme un arbre de manière programmatique via l'API DOM. Nous allons voir ces deux approches dans ce chapitre, en mettant l'emphase sur la proximité avec les grammaires attribuées ou en utilisant ces dernières pour définir des traitements efficaces sur les documents.

8.1 Le langage de transformation XSL

La langage de transformation XSL (XML stylesheet language) est un langage permettant de programmer des transformations de documents (changements de structure) ou des traductions de documents vers d'autres formats (texte, HTML, etc.). XSL a été d'abord défini comme un langage permettant de traduire un document XML en une forme imprimable ou présentable à un utilisateur, d'où son nom de langage de feuilles de styles. Pourtant, les créateurs de XSL se sont vite rendus compte de la puissance et de la portée beaucoup plus large d'un tel langage de transformation.

Aujourd'hui, la traduction d'un document XML vers une forme imprimable est plutôt réalisée à l'aide des CSS («*Cascading StyleSheet*»), une norme concurrente développée spécifiquement pour cela. XSL est plutôt utilisé pour les transformations de documents, bien qu'une norme (XSL-FO, pour *formatting objects*) permettra bientôt de réaliser des transformations vers des documents formattés avec une grande finesse tout en profitant de toute la puissance de XSL.

Un programme XSL se présente sous la forme d'un ensemble de règles de transformation. Une règle de transformation spécifie par un filtre les nœuds d'une arborescence de document auxquels elle s'applique dans le document source. Elle donne ensuite une forme à générer dans le document cible. On aurait pu définir n'importe quelle syntaxe spécifique pour XSL, mais dans l'esprit de XML, il était intéressant de définir les programmes XSL comme des documents XML. XSL est donc un langage à balises que l'on peut définir par une DTD. Une règle XSL se présente donc selon la forme suivante :

```
<xsl:template match="filtre" >
  forme
</xsl:template>
```

Lors du parcours d'un document par le moteur de transformation, le filtre déclenche l'application d'une règle lorsqu'un nœud rencontré correspond au filtre. Le filtre est une expression selon la norme XPATH. Cette norme permet de désigner un nœud dans un arbre XML en fonction de son type d'élément, la présence d'un certain attribut ou de la valeur d'un attribut, des nœuds frères, ascendants ou descendants (type d'élément, attributs ou valeurs, etc.).

La partie filtre de la règle

XPATH est un langage très puissant que nous n'allons pas étudier en détail. Une expression XPATH est composée :
- de *termes de positionnement*, qui définissent une position relative au nœud courant, comme par exemple les fils d'un nœud par `child` ;
- de *filtres* introduits par l'opérateur '`::`', qui précisent des caractéristiques d'un nœud, comme par exemple le nom de l'élément ; puis
- de prédicats optionnels présenté entre crochets qui sont des expressions logiques formées sur un langage de prédicats spécifiquement défini pour XSL, comme par exemple la position d'un élément fils par rapport à l'ensemble des fils d'un nœud.

Exemple 8.1 L'expression XPATH `child::chapitre[position()=3]` sélectionne un nœud ayant un élément fils (`child`) de nom `chapitre` dont la position parmi les autres fils est 3. □

Les principaux indicateurs de position relative XPATH sont : `child`, `descendant`, `parent`, `ancestor`, `preceding`, `following` et `attribute`. D'autres existent. Le filtre XPATH peut être un nom d'élement, un nom d'attribut, le passe-partout '`*`' (pour tout objet de même nature), un nom d'élément ou d'attribut associé à un prédicat entre crochets. Les prédicats sont des expressions dans lesquelles on peut utiliser les opérateurs logiques `AND` et `OR`, les opérateurs de comparaison `<`, `>`, `<=`, `>=`, `=` et `!=`. On peut aussi utiliser des expressions arithmétiques et des fonctions prédéfinies (comme `position()`).

Notons également que la partie filtre d'une règle de transformation peut être composée de plusieurs expressions XPATH séparées par des barres verticales '`|`' qui indiquent l'alternative. La règle s'applique alors à toutes les alternatives filtrées.

La partie forme de la règle

La partie *forme* de la règle est constituée d'instructions du langage XSL pour construire la résultat de la transformation correspondant au nœud sélectionné par le filtre de la règle. Ces instructions comportent des instructions permettant la création de nœuds et d'attributs (comme `xsl:element` et `xsl:attribute`), des instructions permettant de récupérer le contenu de l'élément source (comme `xsl:value-of`) et des instructions permettant d'appliquer récursivement les règles de transformation du programme XSL sur les nœuds fils du nœud selectionné (comme `xsl:apply-templates`). XSL définit également des structures de contrôle : alternatives (`xsl:if` et `xsl:choose`) et répétition (`xsl:for-each`).

Absence de règle de transformation

Lorsqu'il n'y a pas de règle de transformation qui s'applique à un nœud du document source, XSL applique l'une des deux règles par défaut suivantes :

R1 : Les règles de transformation sont appliquées récursivement à tous les descendants du nœud.

R2 : Si le nœud courant contient du texte de type `#PCDATA`, alors ce texte est recopié tel quel dans le document cible.

Structure d'une feuille XSL

Comme tout document XML, une feuille XSL comporte une entête formée d'éléments donnant des informations sur l'ensemble du document. Cette entête contient l'instruction XSL qui indique qu'il s'agit d'un document se conformant à la norme XSL, dont la forme est :

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
```

Elle peut aussi comporter un élément `xsl:output` dont le rôle consiste à préciser le format de sortie de la transformation (type de document, XML, HTML, ou texte, DTD s'il s'agit d'un document XML, etc.).

Après cette entête, on trouve un ensemble de règles données par des éléments `xsl:template`, puis la balise fermante de l'élément `xsl:stylesheet`, c'est-à-dire `</xsl:stylesheet>`.

Principales instructions XSL

Les principales instructions XSL sont les suivantes, avec les attributs qui leurs sont attachés :

- `<xsl:apply-templates>` : applique récursivement les règles de transformation aux fils du nœud courant. Elle peut avoir un attribut `select` pour restreindre l'ensemble des nœuds auxquels elle s'applique.
- `<xsl:value-of>` : produit la valeur de l'élément ou de l'attribut auquel on l'applique. Son comportement varie en fonction de l'objet auquel on l'applique ; pour un élement ou un attribut, elle retourne une chaîne de caractère représentant le contenu ou la valeur.
- `<xsl:text>` : reproduit son contenu dans le document cible.
- `<xsl:copy>` : copie l'élément courant du document source dans l'arborescence du document cible (mais pas les fils, il s'agit de copie superficielle).
- `<xsl:element name="`*n*`">` : crée un élément de nom *n* dans l'arborescence du document cible.
- `<xsl:attribute name="`*n*`">` : à l'intérieur d'une instruction `xsl:element`, ajoute à cet élément un attribut de nom *n* dont la valeur est donné par l'évaluation du contenu de l'instruction.
- `<xsl:if test="`*exp*`">` : si l'expression booléenne *exp* est vraie, alors le contenu de l'instruction est exécuté, sinon rien n'est fait.
- `<xsl:choose>` : exécute l'une ou l'autre des instruction d'alternatives qu'elle contient, à la manière d'une instruction `case` généralisée. Les instructions des alternatives peuvent être des formes suivantes :
  `<xsl:when test="`*exp*`">` : le contenu de l'élément est exécuté si *exp* est vraie.
  `<xsl:otherwise>` : si le contrôle parvient à cet élément, son contenu est exécuté par défaut.
- `<xsl:for-each select="`*exp*`">` : *exp* est une expression XPATH qui doit produire une liste de nœuds à chacun desquels le contenu de l'instruction est appliqué.

Exemple 8.2 Voici un exemple tiré du livre de Michard [Mic01]. Considérons un document accumulant le noms, adresses et numéros de téléphone et télécopie de personnes travaillant dans une entreprise :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
```

```
<!DOCTYPE cartes [
  <!ELEMENT cartes (personne+) >
  <!ELEMENT personne (nom, prenom, bureau) >
  <!ELEMENT nom (#PCDATA) >
  <!ELEMENT prenom (#PCDATA) >
  <!ELEMENT bureau (compagnie, adresse, bureau-nbr, telephone, telecopie) >
  <!ELEMENT compagnie (#PCDATA) >
  <!ELEMENT adresse (rue, nbr?, post-code, ville, pays) >
  <!ELEMENT rue (#PCDATA) >
  <!ELEMENT nbr (#PCDATA) >
  <!ELEMENT post-code (#PCDATA) >
  <!ELEMENT ville (#PCDATA) >
  <!ELEMENT pays (#PCDATA) >
  <!ELEMENT bureau-nbr (#PCDATA) >
  <!ELEMENT telephone (#PCDATA) >
  <!ELEMENT telecopie (#PCDATA) >
]>
<cartes>
  <personne>
    <nom>Dupond</nom><prenom>Jean</prenom>
    <bureau>
      <compagnie>INRIA</compagnie>
      <adresse>
        <rue>Route des Lucioles</rue>
        <nbr></nbr>
        <post-code>F-06560</post-code>
        <ville>Sophia Antipolis</ville>
        <pays>France</pays>
      </adresse>
      <bureau-nbr>2506</bureau-nbr>
      <telephone>+33.4.9365.7777</telephone>
      <telecopie>+33.4.9365.7788</telecopie>
    </bureau>
  </personne>
  <personne>
    <nom>Michard</nom><prenom>Alain</prenom>
    <bureau>
      <compagnie>INRIA</compagnie>
      <adresse>
        <rue>Domaine de Voluceau</rue>
        <nbr>BP 105</nbr>
        <post-code>F-78153</post-code>
        <ville>Le Chesnay Cedex</ville>
        <pays>France</pays>
      </adresse>
      <bureau-nbr>3276</bureau-nbr>
      <telephone>+33.1.3963.7777</telephone>
      <telecopie>+33.1.3963.5114</telecopie>
    </bureau>
  </personne>
```

```
</cartes>
```

On souhaite produire deux présentations différentes de ce document. La première présentation est un annuaire téléphonique classique destiné à être imprimé sur papier et qui comportera toutes les informations connues sur chacune des personnes. La seconde présentation est une liste de noms avec compagnies, numéros de téléphone et de télécopie destinée plutôt à être consultée en ligne via un navigateur internet. Cet exemple illustre donc plusieurs utilisations d'un même document de base.

Pour produire la première présentation, on utilise la transformation suivante :

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='yes' ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"
            doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"
            indent="yes" encoding="ISO-8859-1"
            omit-xml-declaration="no" />
<!-- ************************************************************ -->
<xsl:template match="/">
  <HTML>
  <HEAD>
  <TITLE>Annuaire téléphonique</TITLE>
  <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
  </HEAD>
  <BODY>

  <xsl:apply-templates />

  </BODY>
  </HTML>
</xsl:template>
<!-- ************************************************************ -->
<xsl:template match="personne">
  <p>
  <xsl:apply-templates select="prenom" />
  <xsl:text> </xsl:text>
  <xsl:apply-templates select="nom" />
  <br/>
  <xsl:apply-templates select="bureau" />
  </p>
</xsl:template>
<!-- ************************************************************ -->
<xsl:template match="nom">
  <b><xsl:value-of select="." /></b>
</xsl:template>
<!-- ************************************************************ -->
<xsl:template match="prenom">
  <xsl:value-of select="." />
</xsl:template>
```

```
<!-- ****************************************************************** -->
<xsl:template match="bureau">
  <xsl:apply-templates />
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="compagnie">
  <xsl:value-of select="." /><br/>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="adresse">
  <xsl:apply-templates />
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="rue">
  <xsl:value-of select="." /><br/>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="nbr">
  <xsl:if test="string-length() > 0">
    <xsl:value-of select="." /><br/>
  </xsl:if>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="post-code">
  <xsl:value-of select="." /><xsl:text> </xsl:text>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="ville">
  <xsl:value-of select="." /><xsl:text> </xsl:text>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="pays">
  <xsl:value-of select="." /><br/>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="bureau-nbr">
  <em>bureau n° : </em><xsl:value-of select="." /><br/>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="telephone">
  <em>téléphone : </em><xsl:value-of select="." /><br/>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="telecopie">
  <em>télécopie : </em><xsl:value-of select="." />
</xsl:template>
<!-- ****************************************************************** -->
</xsl:stylesheet>
```

L'élément `xsl:output` indique que le document cible est un document HTML dont le type est donné par une référence publique. Les autres attributs indiquent que l'indentation des éléments doit être préservée, que l'encodage est ISO Latin1 et que le document cible doit inclure toutes les déclarations XML.

Outre la première règle, dont le filtre `"/"` s'apparie par définition à la racine du document source, les filtres contiennent simplement le nom des éléments auxquels la règle doit s'appliquer. La forme de la première règle inclut dans le document cible. Un appel à `xsl:apply-templates` applique récursivement les règles aux fils de la racine.

La règle sur les éléments de nom `personne` utilise l'instruction `xsl:text` pour produire un texte contrôlé explicitement par le fait qu'on doit reproduire exactement tout le contenu de cet élément (ici un caractère espace). L'utilisation d'appels à `xsl:apply-templates` avec un sélecteur particulier contrôle l'ordre dans lequel les fils vont être traités et permet d'insérer des formes à générer entre les résultats sur les fils. L'instruction `xsl:value-of` avec un sélecteur égal à un point s'applique au nœud courant filtré par la règle. L'utilisation d'une instruction `xsl:if` dans la règle s'appliquant aux éléments `nbr` permet de ne générer un texte que si cet élément dans le document source n'est pas vide ; on voit ici l'utilisation d'une fonction prédéfinie `string-length` s'appliquant au contenu de l'élément courant.

La transformation suivante produit un tableau HTML avec uniquement les noms, prénoms, la compagnie et les numéros de téléphone et de télécopie :

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='yes' ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"
            doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"
            indent="yes" encoding="ISO-8859-1"
            omit-xml-declaration="no" />
<!-- ************************************************************* -->
<xsl:template match="/">
  <HTML>
  <HEAD>
  <TITLE>Annuaire complet</TITLE>
  <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
  </HEAD>
  <BODY>

  <table border="1">
  <tr>
  <td>Prénom</td>
  <td>Nom</td>
  <td>Compagnie</td>
  <td>Téléphone</td>
  <td>Télécopie</td>
  </tr>
  <xsl:apply-templates />
  </table>

  </BODY>
  </HTML>
```

```
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="personne">
  <tr>
  <xsl:apply-templates select="prenom" />
  <xsl:apply-templates select="nom" />
  <xsl:apply-templates select="bureau" />
  </tr>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="nom">
  <td><xsl:value-of select="." /></td>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="prenom">
  <td><xsl:value-of select="." /></td>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="bureau">
  <xsl:apply-templates />
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="compagnie">
  <td><xsl:value-of select="." /></td>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="adresse">
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="bureau-nbr">
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="telephone">
  <td><xsl:value-of select="." /></td>
</xsl:template>
<!-- ****************************************************************** -->
<xsl:template match="telecopie">
  <td><xsl:value-of select="." /></td>
</xsl:template>
<!-- ****************************************************************** -->
</xsl:stylesheet>
```

Cette seconde transformation ressemble à la première, à la création du tableau en HTML près. Plusieurs règles ont une forme vide ; ceci permet d'éviter le traitement par défaut appliqué par XSL selon les deux règles vues précédemment. Sans les règles à forme vide, un élément de contenu type `#PCDATA` verrait ce contenu copié dans le document cible. □

8.2 Grammaires attribuées et transformations

Nous avons déjà fait le parallèle entre grammaire non-contextuelle et DTD. Il paraît tout aussi intéressant de faire un parallèle entre les transformations XSLT et les règles sémantiques des grammaires attribuées. En quelque sorte, il est tentant d'écrire :

$$\text{DTD} + \text{transformation XSL} = \text{Grammaire attribuée}$$

En effet, une transformation XSL permet de traiter les nœuds d'un sous-arbre tout en insérant des actions sémantiques entre ceux-ci. On a donc une analogie entre transformations XSL et grammaires L-attribuées. Cependant, il existe plusieurs restrictions à cette équation.

D'abord, il faut comprendre que XSL est une langage d'actions sémantiques beaucoup plus puissant que les grammaires L-attribuées par le fait que l'on peut adresser dans une règle XSL non seulement le nœud courant (c'est-à-dire correspondant au non-terminal en partie gauche de la règle) et les nœuds fils (c'est-à-dire correspondant aux symboles de la partie droite de la production) mais en général tout autre nœud dans l'arbre du document. Bien entendu, cette possibilité a un revers, qui est l'obligation de maintenir tout l'arbre du document accessible en mémoire pendant toute la transformation. Les grammaires attribuées observent un principe de localité dans les règles sémantiques qui permet, comme nous l'avons vu, de parcourir une seule fois l'arbre de dérivation, voire de ne même pas le construire.

Malgré cela, les transformations XSL sont aussi limitées par rapport aux grammaires attribuées. En effet, une transformation XSL ne peut pas modifier l'arbre source de la transformation. Il est possible en XSL de récupérer des portions déjà générées de l'arbre cible, par le mécanisme des variables et des paramètres. Cependant, ces *fragments d'arbre* ont un type spécifique différent du type document XML. En pratique, cela veut dire qu'on ne peut presque rien faire avec un fragment d'arbre, ce type étant totalement opaque. La principale chose que l'on peut faire est de le copier dans le document cible. On utilise cette possiblité lorsqu'on veut générer une seule fois un fragment d'arbre mais l'insérer plusieurs fois dans l'arborescence cible.

La conséquence de la restriction d'XSL à ne pas modifier les arbres et à ne pouvoir accéder au contenu des fragments d'arbres est qu'on ne peut en une seule passe de transformation calculer l'équivalent d'attributs synthétisés ou hérités sur un document XML. Pour réaliser l'équivalent d'un calcul d'attributs, il faut s'en remettre à un autre outil de manipulation des arbres de documents XML : l'API DOM que nous verrons à la section 8.3.

La conclusion des deux remarques précédentes est qu'une transformation XSL, à condition de respecter le principe de localité des règles sémantiques et de ne faire qu'une transformation d'un arbre source en un arbre cible, peut être vue comme un schéma de traduction. Des applications classiques des schémas de traduction comme la construction d'un arbre abstrait ou la re-synthèse d'un source de programme à partir d'un arbre de dérivation (*pretty-print*) sont donc facilement réalisables par une transformation XSL (à ceci près que le contrôle fin du format de sortie est parfois délicat en XSL).

Considérons d'abord le problème de la création d'un arbre abstrait à partir d'un arbre de dérivation, et ce par un exemple concret.

Exemple 8.3 Supposons une grammaire concrète des expressions que nous avons souvent

utilisée :

$$\begin{array}{lcl} E & \rightarrow & E\text{+}T \mid T \\ T & \rightarrow & T*F \mid F \\ F & \rightarrow & (E) \mid N \mid I \end{array}$$

La DTD correspondante, `concrete.dtd`, obtenue selon les règles de la section 5.3 :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ELEMENT E ((E, Plus, T) | T) >
<!ELEMENT T ((T, Fois, F) | F) >
<!ELEMENT F ((pg, E, pd) | N | I) >
<!ELEMENT N (#PCDATA) >
<!ELEMENT I (#PCDATA) >
<!ELEMENT Plus EMPTY >
<!ELEMENT Fois EMPTY >
<!ELEMENT pg EMPTY >
<!ELEMENT pd EMPTY >
```

Une grammaire abstraite suffisante pour exprimer de manière équivalente les phrases reconnues par la grammaire précédente est :

$$e \rightarrow e\text{+}e \mid e*e \mid n \mid i$$

La DTD correspondante est, `abstraite.dtd` :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ELEMENT e ((e, plus, e) | (e, fois, e) | n | i) >
<!ELEMENT n (#PCDATA) >
<!ELEMENT i (#PCDATA) >
<!ELEMENT plus EMPTY >
<!ELEMENT fois EMPTY >
```

Il est possible de définir une définition dirigée par la syntaxe réalisant la construction d'un arbre abstrait à partir de l'arbre de dérivation. L'attribut synthétisé `a` va contenir l'arbre abstrait.

Production	Règle sémantique
$E \rightarrow E_1\text{+}T$	E`.a := creerNoeud(`e`, `E_1`.a, creerNoeud(+), `T`.a)`
$E \rightarrow T$	E`.a := `T`.a)`
$T \rightarrow T_1*F$	T`.a := creerNoeud(`e`, `T_1`.a, creerNoeud(*), `F`.a)`
$T \rightarrow F$	T`.a := `F`.a)`
$F \rightarrow (E)$	F`.a := `E`.a)`
$F \rightarrow N$	F`.a := creerNoeud(`e`, creerNoeud(`n`, `N`.vallex))`
$F \rightarrow I$	F`.a := creerNoeud(`e`, creerNoeud(`i`, `I`.vallex))`

La transformation XSL suivante prend un document respectant la DTD `concrete.dtd` et la transforme en un document sémantiquement équivalent respectant la DTD `abstraite.dtd`.

Elle est obtenue de la définition dirigée par la syntaxe en utilisant la fonction `xsl:element` pour créer les nœuds et la fonction `xsl:apply-templates` pour calculer les valeurs de l'attribut synthétisé `a`. On utilise des structures de contrôle `xsl:choose` pour traiter indépendamment les différents types de nœuds de l'arbre de dérivation (document source) selon la production appliquée.

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='yes' ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"
            doctype-system="abstraite.dtd"
            indent="yes" encoding="ISO-8859-1"
            omit-xml-declaration="no" />
<xsl:preserve-space elements="*" />
<!-- ************************************************************ -->
<xsl:template match="E">
  <xsl:choose>
    <xsl:when test="child::Plus">
      <xsl:element name="e">
        <xsl:apply-templates select="E" />
        <xsl:element name="plus" />
        <xsl:apply-templates select="T" />
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ************************************************************ -->
<xsl:template match="T">
  <xsl:choose>
    <xsl:when test="child::Fois">
      <xsl:element name="e">
        <xsl:apply-templates select="T" />
        <xsl:element name="fois" />
        <xsl:apply-templates select="F" />
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ************************************************************ -->
<xsl:template match="F">
  <xsl:choose>
    <xsl:when test="child::pg">
      <xsl:apply-templates select="E" />
    </xsl:when>
```

```
      <xsl:when test="child::N">
        <xsl:element name="e">
          <xsl:apply-templates select="N" />
        </xsl:element>
      </xsl:when>
      <xsl:otherwise>
        <xsl:element name="e">
          <xsl:apply-templates select="I" />
        </xsl:element>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <!-- *************************************************************** -->
  <xsl:template match="N">
    <xsl:element name="n">
      <xsl:value-of select="." />
    </xsl:element>
  </xsl:template>
  <!-- *************************************************************** -->
  <xsl:template match="I">
    <xsl:element name="i">
      <xsl:value-of select="." />
    </xsl:element>
  </xsl:template>
  <!-- *************************************************************** -->
  </xsl:stylesheet>
```

L'exécution de cette transformation sur la programme suivant :

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='no' ?>
<!DOCTYPE E SYSTEM 'concrete.dtd'>
<E>
  <E>
    <T>
      <T><F><N>4</N></F></T>
      <Fois/>
      <F><N>3</N></F>
    </T>
  </E>
  <Plus/>
  <T>
    <T><F><I>x</I></F></T>
    <Fois/>
    <F>
      <pg/>
        <E>
          <E><T><F><I>y</I></F></T></E>
          <Plus/>
          <T><F><N>8</N></F></T>
```

```
        </E>
      <pd/>
    </F>
  </T>
</E>
```

donnera (aux indentations près) le programme abstrait suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE e SYSTEM "abstraite.dtd">
<e>
  <e>
    <e><n>4</n></e>
    <fois/>
    <e><n>3</n></e>
  </e>
  <plus/>
  <e>
    <e><i>x</i></e>
    <fois/>
    <e>
      <e><i>y</i></e>
      <plus/>
      <e><n>8</n></e>
    </e>
  </e>
</e>
```

□

Par l'exemple précédent, on voit qu'il est facile de transformer une définition dirigée par la syntaxe calculant un attribut synthétisé en une transformation XSL. Remarquons cependant que jamais dans le calcul d'un attribut a-t-on besoin d'accéder au contenu des attributs des nœuds fils. Cette propriété de la définition dirigée par la syntaxe de l'exemple rend possible sa traduction en XSL.

Considérons maintenant l'exemple de la synthèse d'un source sur la grammaire précédente. Dans le cas de la synthèse de source, comme nous l'avons vu au chapitre précédent, il faut utiliser un schéma de traduction avec des actions intérieures aux règles de production.

Exemple 8.4 Considérons à nouveau la grammaire précédente. Comme nous l'avons vu au chapitre précédent, le schéma de traduction suivant réalise la synthèse d'un source de programme à partir de l'arbre de dérivation :

$$
\begin{array}{lcl}
E & \rightarrow & E\texttt{+}\ \{\texttt{Imp('+')}\}\ T \mid T \\
T & \rightarrow & T\texttt{*}\ \{\texttt{Imp('*')}\}\ F \mid F \\
F & \rightarrow & \texttt{(}\ \{\texttt{Imp('(')}\}\ E\texttt{)}\ \{\texttt{Imp(')')}\} \mid N\ \{\texttt{Imp(}N\texttt{.vallex)}\} \mid I\ \{\texttt{Imp(}I\texttt{.vallex)}\}
\end{array}
$$

Ce schéma peut se traduire de la manière suivante en XSL :

```
<?xml version="1.0" encoding='ISO-8859-1' standalone='yes' ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="no" encoding="ISO-8859-1" />
<xsl:strip-space elements="*" />
<!-- ******************************************************************** -->
<xsl:template match="E">
  <xsl:choose>
    <xsl:when test="child::Plus" >
      <xsl:apply-templates select="E" />
      <xsl:text>+</xsl:text>
      <xsl:apply-templates select="T" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ******************************************************************** -->
<xsl:template match="T">
  <xsl:choose>
    <xsl:when test="child::Fois" >
      <xsl:apply-templates select="T" />
      <xsl:text>*</xsl:text>
      <xsl:apply-templates select="F" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ******************************************************************** -->
<xsl:template match="F">
  <xsl:choose>
    <xsl:when test="child::E" >
      <xsl:text>(</xsl:text>
      <xsl:apply-templates select="E" />
      <xsl:text>)</xsl:text>
    </xsl:when>
    <xsl:when test="child::N" >
      <xsl:value-of select="N" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="I" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- ******************************************************************** -->
</xsl:stylesheet>
```

□

8.3 API DOM et la transformation de documents

Si le langage de transformation XSL impose des contraintes telles qu'il est impossible d'y calculer effectivement des attributs au sens des grammaires attribués, l'API DOM donne cette flexibilité. DOM est une interface standardisée qui permet de récupérer l'arborescence d'un document XML sous la forme d'objets manipulables dans un langage de programmation. Plusieurs implantations de cette API existent dans les langages à objets courants, dont Java et Python. La librairie Xerces, par exemple, en donne une implantation pour Java.

8.3.1 Définition de l'API DOM

Vue de manière très générale, l'API DOM telle que définie par le W3C consiste en dix-sept interfaces et une exception (la figure 8.1 en présentent les principales) qui commandent comment accéder aux différents types de nœuds dans un arbre de document XML. Les interfaces `Node` et `Element` définissent les méthodes permettant de naviguer dans l'arborescence, interroger un nœud pour en connaître le type ou encore le nom d'élément, ou encore pour modifier l'arbre, en ajoutant un attribut avec sa valeur à un élément par exemple.

Pour transformer un document XML, il suffit donc de lire ce document en utilisant un analyseur qui retourne un arbre de document respectant l'API DOM, puis de traverser cet arbre pour lui appliquer les modifications attendues, et finalement réécrire l'arbre comme un document XML dans un fichier résultat. La distribution de Xerces contient une classe `DOMWriter` qui illustre comment analyser un document pour en récupérer un arbre, puis comment afficher cet arbre comme un document XML au terminal. Une simple redirection suffit pour en reverser le contenu dans un fichier. Il suffit donc de modifier légèrement cette classe pour intervertir une transformation utilisant DOM entre l'analyse du fichier source et l'écriture du résultat.

Pour illustrer l'utilisation de l'API DOM, considérons le problème de récupérer un tableau des éléments d'un certain nom parmi les nœuds fils d'un certain élément. La méthode `getChildrenElementOfName` suivante réalise cette récupération :

```
protected Element[]          getChildrenElementOfName(
    Element           element,
    java.lang.String name
    )
{
    Node             aChildNode ;
    NodeList         children ;
    Element[]        ret, tmp ;
    int              elementCount, i ;

    children = element.getChildNodes() ;
    int noOfChildren = children.getLength() ;
    tmp = new Element[noOfChildren] ;
    elementCount = 0 ;
    for (i = 0 ; i < noOfChildren ; i++) {
        aChildNode = children.item(i) ;
        if (aChildNode.getNodeType() == Node.ELEMENT_NODE &&
                     ((Element)aChildNode).getTagName().equals(name)) {
            tmp[elementCount++] = (Element)aChildNode ;
```

Interfaces	
Attr	représente un attribut dans un objet `Element`.
CharacterData	étend l'interface `Node` avec un ensemble de méthodes pour accéder à des données de type caractère.
Comment	hérite de `CharacterData` et représente le contenu d'un commentaire.
Document	représente un document XML entier.
DocumentType	représente le type de documents et les informations qu'il contient (nom de la DTD, entités, ...).
DOMImplementation	propose des méthodes pour obtenir de l'information de l'implantation sous-jacente de DOM de manière indépendante de l'implantation.
Element	hérite de `Node` et réprésente un élément dans le document.
Node	représente le type de données le plus général de l'API DOM, un nœud dans l'arbre du document.
NodeList	représente une abstraction sur une collection de nœuds dans un document.
Text	hérite de `CharacterData` et représente le contenu textuel d'un élément ou d'un attribut.
Exceptions	
DOMException	exception levée lorsqu'une opération est impossible.

FIG. 8.1 – Interfaces de l'API DOM du W3C.

```
            } // end of if ()
        } // end of for ()

        ret = new Element[elementCount] ;
        for (i = 0 ; i < elementCount ; i++) {
            ret[i] = tmp[i] ;
        } // end of for ()

        return ret ;

    } // ------------------------------------- getChildrenElementofName()
```

L'interface `Element` nous offre la méthode `getTagName` pour récupérer le nom d'un élément alors que l'interface `Node` nous offre les méthodes :
- `getChildNodes` qui retourne une liste (`NodeList`) des nœuds fils du receveur, et
- `getNodeType()` qui retourne le type de nœud sous la forme d'un entier court (`short`) à comparer aux constantes prédéfinies dans l'interface `Node` ;

Avec ces méthodes, il est facile de récupérer les fils, d'itérer sur ceux-ci de manière à repérer les nœuds éléments du nom donné et de les accumuler dans un tableau (par l'intermédiaire d'un tableau temporaire de manière à créer un tableau dont la taille correspond exactement au nombre d'éléments retournés).

Une tâche a priori un peu plus simple consiste à vérifier s'il existe parmi les nœuds fils d'un élément un qui soit un élément d'un certain nom. En utilisant le même schéma que pour la

méthode précédente, la méthode `hasChildElementOfName` suivante réalise cette vérification en optimisant l'itération sur les nœuds fils de manière à terminer dès qu'un élément fils du bon nom est trouvé :

```
protected boolean          hasChildElementOfName(
    Element          element,
    java.lang.String name
    )
{
    Node            aChildNode ;

    NodeList children = element.getChildNodes() ;
    int noOfChildren = children.getLength() ;
    boolean hasChildElement = false ;
    for (int i = 0 ; !hasChildElement && i < noOfChildren ; i++) {
        aChildNode = children.item(i) ;
        if (aChildNode.getNodeType() == Node.ELEMENT_NODE &&
                     ((Element)aChildNode).getTagName().equals(name)) {
            hasChildElement = true ;
        } // end of if ()
    } // end of for ()

    return hasChildElement ;

} // ----------------------------------------- hasChildElementofName()
```

8.3.2 Parcours de l'arbre du document

L'objectif que nous nous fixons est d'évaluer les attributs d'une grammaire L-attribuée, en supposant que la grammaire a été modifiée pour remplacer les accès aux attributs hérités par des accès dans la pile, comme nous l'avons vu à la fin de la section précédente. Cet ordre d'évaluation correspond à un parcours de l'arbre de gauche à droite en profondeur d'abord. Pour les besoins de l'évaluation des attributs, il faut être capable de conserver les éléments déjà traités dans une pile où il est possible d'accéder aux éléments relativement à l'élément en sommet de pile. La figure 8.2 propose une classe dérivée de la classe standard `java.util.Stack` de la librairie Java pour ce faire.

L'algorithme de parcours devient alors relativement simple. En partant de la racine, il faut traiter les nœuds fils de gauche à droite avant de traiter le nœud parent. Lors du traitement du nœud parent, les nœuds fils sont dépilés, puis on doit traiter le nœud parent avant de l'empiler à son tour dans la pile de nœuds traités.

Plutôt que de donner un algorithme général, considérons la construction d'une transformation par descente récursive illustrée par un exemple précis.

Exemple 8.5 Considérons à nouveau la grammaire abstraite de la section précédente :

$$e \rightarrow e{+}e \mid e{*}e \mid n \mid i$$

```
/**
 * <code>StackWithRelativeAccess</code> extends the standard
 * <code>java.util.Stack</code> class in order to provide an
 * access to stack elements by an index relative to the top
 * element in the stack.
 *
 * <p>Created: Thu May 22 18:10:39 2003</p>
 *
 * @author Jacques Malenfant
 * @version 0.1 -- 05/2003
 */
public class            StackWithRelativeAccess extends java.util.Stack
{
    /**
     * Peeks (accesses without removing) a stack element at
     * <code>index</code> positions under the top of the stack
     * (<code>index</code> == 0 means the top of the stack).
     *
     * @param index     the position from the top to access.
     * @return          the accessed elements, if it exists.
     * @exception java.util.EmptyStackException when there is no
     *                  such element in the stack.
     */
    public Object               peekAt(int index)
        throws java.util.EmptyStackException
    {
        Object          ret ;

        if (elementCount >= index) {
            ret = elementData[elementCount-(index+1)] ;
        } else {
            throw new java.util.EmptyStackException() ;
        } // end of if ()else
        return ret ;

    } // --------------------------------------------------------- peekAt()

} // *************************************** class StackWithRelativeAccess
```

Fig. 8.2 – La classe StackWithRelativeAccess

et la DTD correspondante abstraite.dtd :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ELEMENT e ((e, plus, e) | (e, fois, e) | n | i) >
<!ELEMENT n (#PCDATA) >
<!ELEMENT i (#PCDATA) >
<!ELEMENT plus EMPTY >
```

```
<!ELEMENT fois EMPTY >
```

Une optimisation que font tous les compilateurs dignes de ce nom consiste à évaluer à la compilation, c'est-à-dire statiquement, les expressions constantes. Dans le cas de la grammaire abstraite précédente, toute expression ne comportant que des nombres peut être considérée comme constante. Par contre, une expression qui contient des identificateurs doit être évaluée à l'exécution, c'est-à-dire dynamiquement, puisqu'on ne connaît pas statiquement la valeur des variables.

L'analyse qui cherche à répartir les expressions entre celles qui sont évaluables statique et celles qui le sont dynamiquement est appelée analyse des moments de liaison («*binding-time analysis*»). Ici, elle est très simple, puisque par définition une expression contenant uniquement un nombre est statique, et toute expression d'addition ou de multiplication qui s'applique à deux sous-expressions statiques est elle-même statique. Dans tous les autres cas, les expressions sont dynamiques. La définition dirigée par la syntaxe suivante rend compte de cette idée en exprimant le calcul d'un attribut `ml` (pour moment de liaison) :

Productions	Règles
$e \rightarrow e_1$+e_2	si e_1.ml == *stat* && e_2.ml == *stat* alors e.ml = *stat* sinon e.ml = *dyn*
$e \rightarrow e_1$*e_2	si e_1.ml == *stat* && e_2.ml == *stat* alors e.ml = *stat* sinon e.ml = *dyn*
$e \rightarrow n$	e.ml = *stat*
$e \rightarrow i$	e.ml = *dyn*

L'idée de la programmation de la transformation est la suivante. D'abord, on suppose que la transformation est lancée par la méthode suivante :

```
public void                  transform(
    Document document
    )
{
    this.transformElement(document.getDocumentElement()) ;

} // ------------------------------------------------------ transform()
```

qui récupère l'élément racine du document reçu en paramètre. L'idée est d'écrire une méthode de transformation pour chaque type d'éléments dans le document. On doit d'abord écrire une méthode d'aiguillage qui, étant donné le nom de l'élément, applique la bonne méthode :

```
protected void              transformElement(
    Element         element
    )
{
    if         (element.getTagName().equals("e")) {
        this.transform_e(element) ;
    } else if (element.getTagName().equals("n")) {
        this.transform_n(element) ;
```

```
        } else if (element.getTagName().equals("i")) {
            this.transform_i(element) ;
        } else {
            System.out.println("élément non reconnu : " + element) ;
        } // end of if ()else

    } // ---------------------------------------------- transformElement()
```

ensuite, pour chaque type d'élément on produit une méthode qui vérifie la production utilisée pour construire l'élément et applique la règle sémantique appropriée tout en appliquant la transformation sur les noeuds fils, sans oublier la gestion de la pile des noeuds traités. Pour les éléments de type nombre et identificateur, cela est très simple, puisqu'il suffit de valuer l'attribut et d'empiler le nœud ainsi traité :

```
    protected void              transform_i(
        Element         element
        )
    {
        element.setAttribute("ml", "dyn") ;
        doneStack.push(element) ;

    } // --------------------------------------------------- transform_i()

    protected void              transform_n(
        Element         element
        )
    {
        element.setAttribute("ml", "stat") ;
        doneStack.push(element) ;

    } // --------------------------------------------------- transform_n()
```

Pour les nœuds de type expression, il faut vérifier le cas :

```
    protected void              transform_e(
        Element         element
        )
    {
        Element         aChild, result1, result2 ;
        Element[]       children ;

        if          (this.hasChildElementOfName(element, "plus") ||
                         this.hasChildElementOfName(element, "fois")) {
            children = this.getChildrenElementOfName(element, "e") ;
            this.transformElement(children[0]) ;
            this.transformElement(children[1]) ;
            result2 = (Element)doneStack.pop() ;
```

```
            result1 = (Element)doneStack.pop() ;
            if (result1.getAttribute("ml").equals("stat") &&
                                result2.getAttribute("ml").equals("stat")) {
                element.setAttribute("ml", "stat") ;
            } else {
                element.setAttribute("ml", "dyn") ;
            } // end of if ()else
            doneStack.push(element) ;
        } else if (this.hasChildElementOfName(element, "n")) {
            children = this.getChildrenElementOfName(element, "n") ;
            this.transformElement(children[0]) ;
            result1 = (Element)doneStack.pop() ;
            element.setAttribute("ml", result1.getAttribute("ml")) ;
            doneStack.push(element) ;
        } else {
            children = this.getChildrenElementOfName(element, "i") ;
            this.transformElement(children[0]) ;
            result1 = (Element)doneStack.pop() ;
            element.setAttribute("ml", result1.getAttribute("ml")) ;
            doneStack.push(element) ;
        } // end of if ()else

    } // ---------------------------------------------------- transform_e()
```

La DTD de la grammaire abstraite est étendue pour tenir compte de cet attribut sur les éléments :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ENTITY % att.ml "ml (stat | dyn) 'dyn'" >
<!ELEMENT e ((e, plus, e) | (e, fois, e) | n | i) >
<!ATTLIST e %att.ml; >
<!ELEMENT n (#PCDATA) >
<!ATTLIST n %att.ml; >
<!ELEMENT i (#PCDATA) >
<!ATTLIST i %att.ml; >
<!ELEMENT plus EMPTY >
<!ELEMENT fois EMPTY >
```

Enfin, sur un programme abstrait défini par le document XML suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE e SYSTEM "abstraite-ml.dtd">
<e>
  <e>
    <e><n>4</n></e>
    <fois/>
    <e><n>3</n></e>
```

```
    </e>
    <plus/>
    <e>
      <e><i>x</i></e>
      <fois/>
      <e>
        <e><i>y</i></e>
        <plus/>
        <e><n>8</n></e>
      </e>
    </e>
  </e>
```

on obtient le résultat :

```
  <?xml version="1.0" encoding="UTF-8"?>
  <e ml="dyn">
    <e ml="stat">
      <e ml="stat"><n ml="stat">4</n></e>
      <fois></fois>
      <e ml="stat"><n ml="stat">3</n></e>
    </e>
    <plus></plus>
    <e ml="dyn">
      <e ml="dyn"><i ml="dyn">x</i></e>
      <fois></fois>
      <e ml="dyn">
        <e ml="dyn"><i ml="dyn">y</i></e>
        <plus></plus>
        <e ml="stat"><n ml="stat">8</n></e>
      </e>
    </e>
  </e>
```

où on peut constater que toutes les expressions ne contenant que des nombres sont désignées comme statiques et les autres comme dynamiques. Une transformation exécutant les expressions statiques peut alors être appliquée pour optimiser le programme. □

On retiendra de l'exemple précédent l'approche générale pour la construction des méthodes de transformation pour chaque type d'élément :

- aiguiller la transformation en fonction des fils de l'élément à transformer,
- appliquer récursivement la transformation sur les éléménts fils,
- intercaler les actions sémantiques pour l'évaluation des attributs, et
- dépiler les éléments fils de la pile des éléments traités pour empiler l'élément père lorsque le dernier fils a été traité et que les attributs de l'élément père ont été calculés.

L'exemple de l'analyse des moments de liaison que nous venons de présenter illustre un calcul d'attribut progressif sur l'arbre du document XML. Par contre, l'attribut moment de

liaison est synthétisé. Il est donc possible de le calculer sans accéder la pile sous le sommet. Considérons maintenant l'exemple du typage déjà abordé au chapitre précédent pour illustrer l'accès à la pile pour remplacer un accès à un attribut hérité obtenu par simple recopie.

Exemple 8.6 La DTD correspondant à la grammaire attribuée de l'exemple 7.6 s'énonce comme suit :

```
<?xml version="1.0" encoding='ISO-8859-1' ?>
<!ENTITY % att.type "type (reel|entier) #IMPLIED" >
<!ELEMENT D (T, L) >
<!ATTLIST D %att.type; >
<!ELEMENT T (#PCDATA) >
<!ATTLIST T %att.type; >
<!ELEMENT L (i | (L, vg, I)) >
<!ATTLIST L %att.ml; >
<!ELEMENT I (#PCDATA) >
<!ATTLIST I %att.ml;
            entree CDATA #REQUIRED >
<!ELEMENT reel EMPTY >
<!ELEMENT entier EMPTY >
<!ELEMENT vg EMPTY >
```

Plutôt que d'utiliser une table des ymboles pour mémoriser le type des identificateurs, nous ajoutons un attribut type aux éléments identificateur qui va être valué par la transformation. Considérons la phrase '`reel id1, id2, id3`' sous la forme d'un document XML :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE D SYSTEM "types.dtd">
<D>
  <T><reel/></T>
  <L>
    <L>
      <L>
        <I entree="1">id1</I>
      </L>
      <vg/>
      <I entree="2">id2</I>
    </L>
    <vg/>
    <I entree="3">id3</I>
  </L>
</D>
```

La transformation obtenue à partir des règles sémantiques de la grammaire attribuée se construit comme la précédente transformation. D'abord, on écrit la méthode d'aiguillage chargée d'appeler les méthode de transformations correspondant à chacun des types d'élements :

```
    protected void              transformElement(
```

```
        Element         element
        )
    {
        if        (element.getTagName().equals("D")) {
            this.transform_D(element) ;
        } else if (element.getTagName().equals("T")) {
            this.transform_T(element) ;
        } else if (element.getTagName().equals("L")) {
            this.transform_L(element) ;
        } else if (element.getTagName().equals("I")) {
            this.transform_I(element) ;
        } else {
            System.out.println("élément non reconnu : " + element) ;
        } // end of if ()else

    } // ----------------------------------------------- transformElement()
```

La méthode de transformation des éléments de type `D` applique récursivement la transformation sur ses deux éléments fils de type `T` et `L` :

```
    protected void              transform_D(
        Element         element
        )
    {
        Element         aChild, result1, result2 ;
        Element[]       children ;

        children = this.getChildrenElementOfName(element, "T") ;
        this.transformElement(children[0]) ;
        children = this.getChildrenElementOfName(element, "L") ;
        this.transformElement(children[0]) ;
        result2 = (Element)doneStack.pop() ;
        result1 = (Element)doneStack.pop() ;
        doneStack.push(element) ;

    } // ---------------------------------------------------- transform_D()
```

La transformation sur `T` est similaire aux transformations de l'exemple précédent puisque sur les éléments `T`, l'attribut type est synthétisé. Il suffit donc de valuer l'attribut en fonction du type de l'élément fils (réel ou entier) :

```
    protected void              transform_T(
        Element         element
        )
    {
        Element         aChild, result1, result2 ;
        Element[]       children ;
```

```
    if        (this.hasChildElementOfName(element, "reel")) {
        element.setAttribute("type", "reel") ;
        doneStack.push(element) ;
    } else {
        element.setAttribute("type", "entier") ;
        doneStack.push(element) ;
    } // end of if ()else

} // ----------------------------------------------------- transform_T()
```

La méthode de transformation des éléments de type `L` concentre les notions importantes ici :

```
protected void              transform_L(
    Element         element
    )
{
    Element         aChild, result1, result2 ;
    Element[]       children ;

    if        (this.hasChildElementOfName(element, "vg")) {
        children = this.getChildrenElementOfName(element, "L") ;
        this.transformElement(children[0]) ;
        children = this.getChildrenElementOfName(element, "I") ;
        this.transformElement(children[0]) ;
        ((Element)doneStack.peek()).setAttribute(
            "type",
            ((Element)doneStack.peekAt(2)).getAttribute("type")) ;
        result2 = (Element)doneStack.pop() ;
        result1 = (Element)doneStack.pop() ;
        doneStack.push(element) ;
    } else {
        children = this.getChildrenElementOfName(element, "I") ;
        this.transformElement(children[0]) ;
        ((Element)doneStack.peek()).setAttribute(
            "type",
            ((Element)doneStack.peekAt(1)).getAttribute("type")) ;
        result1 = (Element)doneStack.pop() ;
        doneStack.push(element) ;
    } // end of if ()else

} // ----------------------------------------------------- transform_L()
```

Lorsque la production utilisée est $L \rightarrow I$ alors, la valeur du type à attribuer à l'identificateur est celui de l'élément de type `T` qui se trouve immédiatement sous le sommet de la pile juste avant de dépiler l'élément `I` pour empiler l'élément `L`. Lorsque la production utilisée est $L \rightarrow L, I$, alors la valeur du type à attribuer à l'identificateur est celui du deuxième élément sous le sommet de la pile, de type `T`, juste avant de dépiler les éléments `I` et `L` pour empiler l'élément `L` parent. Les utilisations de `peek` et `peekAt` illustrent les accès à la pile comme nous les avons

vu au chapitre précédent (légèrement modifiés ici car nous avons choisi de ne pas empiler l'élément de type `vg`, inutile dans ce contexte). Nous aurions pu modifier les choses en dépilant les éléments fils courants avant d'évaluer la règle sémantique.

Enfin, dans ce contexte, la transformation des éléments de type `I` consiste simplement à les empiler :

```
protected void              transform_I(
    Element         element
    )
{
    doneStack.push(element) ;

} // ---------------------------------------------------- transform_I()
```

L'application de la transformation sur l'exemple précédent retourne le document suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<D>
  <T type="reel"><reel/></T>
  <L>
    <L>
      <L>
        <I entree="1" type="reel">id1</I>
      </L>
      <vg/>
      <I entree="2" type="reel">id2</I>
    </L>
    <vg/>
    <I entree="3" type="reel">id3</I>
  </L>
</D>
```

□

Bibliographie

[ASU89] A. Aho, R. Sethi, et J. Ullman. *Compilateurs — Principes, techniques et outils*. InterÉditions, 1989.

[Fri02] J. Friedl. *Mastering Regular Expressions*. O'Reilly, 2002.

[HU79] J. Hopcroft et J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Mic01] A. Michard. *XML — langage et applications*. Eyrolles, 2001.

[Pera] Perldoc.com. *perlsyn – Perl syntax*. `http://www.perldoc.com/perl5.6/pod/perlsyn.html`.

[Perb] Perldoc.com. *prelre – Perl regular expressions*. `http://www.perldoc.com/perl5.6/pod/perlre.html`.

[Pyt] Site internet python. `http://www.python.org/`.

[Sta94] T.A. Standish. *Data Structures, Algorithms, and Software Principles*. Addison-Wesley, 1994.