111

Théorie des langages et compilation

Octobre 2005
Version 1.0

D. Herman

THÉORIE DES LANGAGES ET COMPILATION

Version 1.0 - 3 octobre 2005

D. HERMAN

111

Théorie des langages et compilation

Daniel HERMAN
Université de Rennes 1 - Ifsic

Avant-propos

Cet ouvrage a été initialement conçu comme support de l'initiation à la compilation pour les étudiants de deuxième année du diplôme d'ingénieur de l'Ifsic[1], le Diic[2].

L'ambition du cours est de donner un socle de connaissances en théorie des langages et en compilation qui soit suffisamment complet pour donner un bagage à tout « honnête informaticien » tout en permettant des approfondissements ultérieurs. Ce socle minimum, mais complet, est actuellement utilisé en mise à niveau pour certains étudiants (formation continue) du Master Informatique[3].

Plusieurs choix stratégiques ont été faits. Le premier est d'ordre pédagogique: l'étudiant est mis en situation d'apprentissage et il doit, à la fois pour apprendre et pour valider ses aptitudes, modifier et étendre un compilateur existant qui lui est fourni. Un certain nombre de connaissances « déclaratives » lui sont toutefois indispensables pour mener a bien les extensions qui lui sont demandées: ces connaissances font l'objet de cet ouvrage. Il faut toutefois être conscient du fait que la *substantifique moelle* de cet enseignement réside dans la réalisation logicielle demandée.

1. L'Institut de Formation Supérieure en Informatique et Communication (Ifsic) est une composante de l'université de Rennes 1.

2. Diic: Diplôme d'Ingénieur en Informatique et Communication.

3. Master dit « pro », 2^{e} année

Un autre choix mérite d'être expliqué: nous avons choisi une analyse syntaxique LL(1) et nous n'utilisons pas de générateur d'analyseur. En fait, nous souhaitons montrer à l'étudiant les principes sous-jacents aux générateurs d'analyseurs et, pour ce faire, l'analyse LL(1) nous semble plus intuitive, plus simple, et cependant suffisante pour mettre en valeur les points importants.

Daniel HERMAN
Rennes, le 1er septembre 2000.
Rennes, le 7 juin 2005.

Plan du cours

- Introduction à la théorie des langages et à la compilation 5
- Automates à nombre fini d'états et langages rationnels 17
- Grammaires et langages 35
- Grammaires algébriques 47
- Analyse LL(1) 57
- Évaluation d'expressions arithmétiques et logiques 77
- Analyse sémantique et génération 91

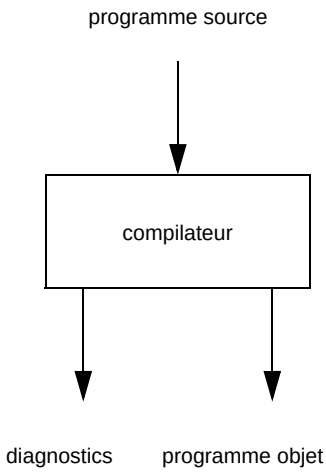
CHAPITRE 1

Introduction à la théorie des langages et à la compilation

1.1 Compilation

1.1.1 Définitions et terminologie

Un *compilateur* est un programme qui, à partir d'un texte de programme, le *programme source*, cherche à produire un programme « équivalent », le *programme objet*, accompagné d'un certain nombre de *diagnostics*.



Les programmes source et objet sont écrits dans des langages également qualifiés respectivement de source et d'objet et ils sont sensés « faire la même chose ». Les diagnostics décrivent des propriétés du programme source et sont particulièrement importants lorsque celui-ci est incorrect: la pertinence des messages d'erreurs est une des qualités agréables d'un compilateur.

Remarque : la terminologie varie selon les auteurs : on utilise indifféremment les mots texte, programme ou code et les termes cible (en anglais *target*) et objet sont synonymes.

Pour espérer avoir des programmes qui « font la même chose » il est important que les langages source et objet soient *rigoureusement* définis. La définition d'un langage comporte deux pans :

- la *syntaxe* qui décrit la forme des constructions autorisées ;
- la *sémantique* qui précise le sens des constructions syntaxiquement correctes.

Pour définir un langage on dispose d'outils conceptuels fournis par un domaine des mathématiques appelé *théorie des langages* ; pour concevoir et réaliser un compilateur on utilise des concepts, des techniques et des outils d'un domaine de l'informatique appelé *compilation*. La compilation utilise évidemment la théorie des langages.

1.1.2 Intérêt

Les enseignements de théorie des langages et de compilation sont presque systématiquement présents dans les cursus de formation supérieure en informatique. On peut trouver au moins quatre explications différentes à cet engouement.

1. La première explication, la plus évidente, est peut-être la moins fondée : si certains étudiants peuvent éventuellement être amenés à participer à la création ou à la maintenance d'un compilateur, leur nombre n'est pas assez important pour que cette raison justifie à elle seule l'intérêt de cet enseignement.
2. Un compilateur est un logiciel complexe dont la production est relativement bien maîtrisée. À ce titre, l'étude des compilateurs relève du génie logiciel : la structuration d'un compilateur est exemplaire, l'outillage conceptuel a permis de mécaniser (générateurs divers) en partie le processus, les impératifs de réutilisation de code (génération multi-cibles…) sont l'objet d'un soin particulier etc.
3. De nombreuses applications comportent de « petits » traducteurs sur lesquels on peut appliquer tout ou partie des principes valables pour ces « gros » traducteurs que sont les compilateurs.
4. Tous les programmes (ou presque) lisent des données et dans les interfaces actuelles on propose d'ordinaire un format souple : les utilisateurs ne tolèrent plus aujourd'hui les contraintes de saisie que pouvaient imposer les informaticiens des années 70 ! De nombreux modules logiciels ont donc pour objet de transcrire un texte résultant d'une saisie en une représentation interne sur laquelle on applique les traitements qui font le « cœur » de l'application. Ces modules réalisent des traductions et leur production peut être

en partie systématisée en utilisant des outils, méthodes ou techniques de la compilation.

Le dernier point nous paraît fondamental.

1.1.3 Arbres abstraits

Un *arbre de syntaxe abstraite*, ou *arbre abstrait* pour faire court, est une description arborescente d'un texte qui en retient les éléments pertinents et les relations qui les lient en se débarrassant des aspects syntaxiques arbitraires. Nous nous limitons à cette vision intuitive qui n'est pas une définition au sens scientifique du terme et nous nous contentons de l'illustrer par des exemples.

Exemple 1

Les deux textes qui suivent sont des conditionnelles rédigées respectivement en Pascal et en C.

```
if E
    then A := 1
    else B := C + 2

if (E)   A = 1 ;
    else B = C + 2 ;
```

L'information contenue dans ces deux textes peut être résumée par l'arbre abstrait suivant:

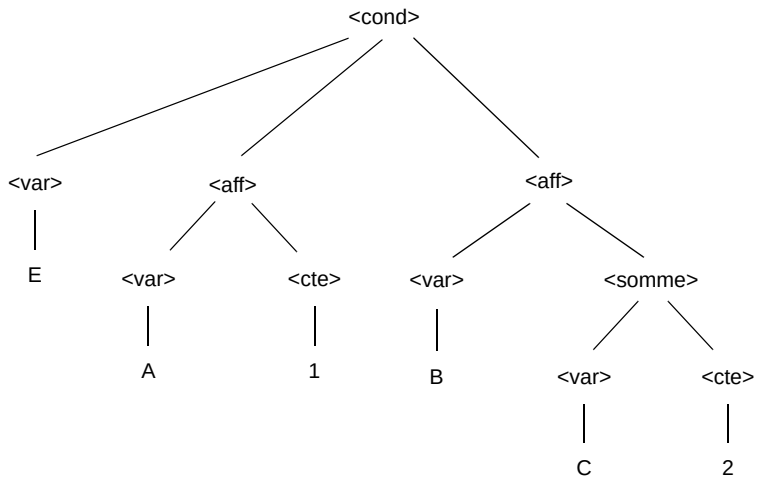


Figure1.1 Arbre abstrait

On peut structurer de nombreuses applications informatiques traitant des données textuelles en un « cœur » qui effectue divers traitements algorithmiques

sur des arbres abstraits et deux interfaces chargées de lire un texte et de le convertir en arbre de syntaxe abstraite ou de réaliser l'opération inverse.

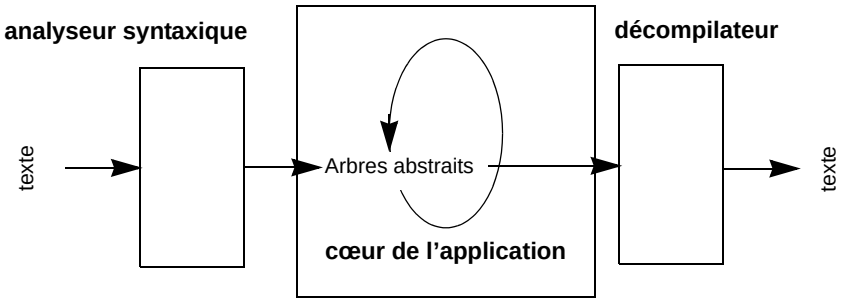


Figure 1.2 Structure générale d'une application traitant des données textuelles

Les textes en entrée et en sortie sont rédigés dans des langages dont il faut donner une description précise, soit pour documenter le logiciel, soit pour concevoir l'application.

Les avantages de la structure présentée ci-dessus sont les suivants:

- La théorie des langages fournit une base conceptuelle et éventuellement des outils de production qui réduisent considérablement les coûts de production des modules « analyseur syntaxique » et « décompilateur ».
- La définition rigoureuse des arbres abstraits manipulés facilite la conception du « cœur » de l'application.

Un compilateur est un cas particulier d'application de ce type, pour lequel on dispose en outre d'une structuration désormais classique pour le « cœur » de l'application.

1.2 Structure d'un compilateur

Pour présenter la structuration d'un compilateur nous empruntons l'exemple proposé dans *Le Dragon*[1]. On considère un fragment de programme source (on suppose que les variables position, initiale et vitesse ont été déclarées et sont de type réel) dont on va suivre le traitement.

Texte source

```
position := initiale +
```

1. Alfred V. AHO, Ravi SETHI, Jeffrey D. ULLMAN, *Compilateurs: principes, techniques et outils*. InterEditions (édition originale 1986, traduction 1989)

```
vitesse * 60
```

1.2.1 Analyse lexicale

L'analyseur lexical repère, dans le texte source, des éléments significatifs appelés *unités lexicales* — ponctuations, mots-clés, constantes, identificateurs — sans tenir compte de leurs liens mutuels; la structure de lignes, les espacements, souvent les commentaires, sont perdus.

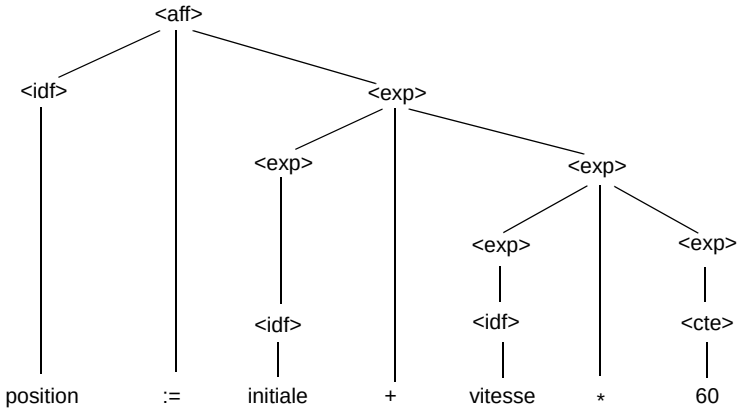
Sortie de l'analyse lexicale

```
<position,identificateur>
<:=,      symb-affectation>
<initiale,identificateur>
<+,       symb-plus>
<vitesse,identificateur>
<*,       symb-mult>
<60,      constante-entière>
```

1.2.2 Analyse syntaxique

L'analyseur syntaxique structure la suite d'unités lexicales en catégories « grammaticales »; on présente d'ordinaire le résultat de cette structuration à l'aide d'un arbre, l'arbre syntaxique.

Sortie de l'analyse syntaxique



1.2.3 Analyse sémantique

L'analyse sémantique a un double but:

- produire un arbre bstrait représentant la « quintessence » du programme;
- effectuer un certain nombre de contrôles dits *statiques* sur la validité du programme.

Les qualificatifs *statique* et *dynamique* s'appliquent, respectivement, à des opérations réalisées avant l'exécution ou pendant l'exécution d'un programme. Les

opérations statiques ne sont donc effectuées qu'une seule fois, alors qu'on peut exécuter un programme des millions de fois… La détection statique des erreurs est donc un enjeu économique majeur.

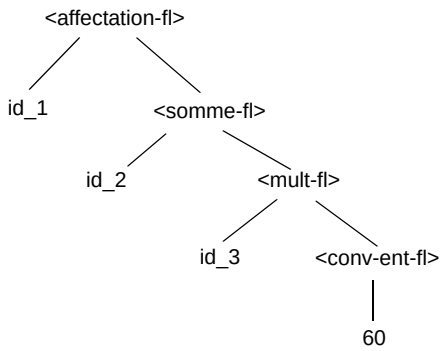
Parmi les contrôles statiques classiques, il faut accorder une place prépondérante au contrôle de type: on s'assure que l'usage des identificateurs et des constantes est conforme, du point de vue de leur type, à ce qui a été annoncé et à l'usage qu'on en fait.

Tout compilateur d'un langage utilisant des variables construit une structure de données, la table des symboles, associant aux identificateurs du programme diverses informations dont leur type.

Pour l'exemple qui nous intéresse, on suppose qu'au moment de l'analyse sémantique la table des symboles contient les informations suivantes:

Identificateur	**Informations associées**	
	type	adresse
…		
position	réel	id_1
initiale	réel	id_2
vitesse	réel	id_3
…		

Sortie de l'analyse sémantique



Du point de vue des types, le fragment analysé est correct; il faut noter toutefois la conversion de l'entier 60 vers le réel 60.0 et le fait que les opérations étiquetant l'arbre abstrait sont des opérations flottantes.

1.2.4 Génération de code

L'arbre abstrait issu de l'analyse sémantique est une idéalisation du programme susceptible d'être adaptée, modulo un coût qu'on espère raisonnable, à diverses machines ou langages cibles. La génération de code prend évidemment en compte la cible finale, mais on distingue souvent trois étapes dans la production du code, les deux premières ne faisant que des hypothèses très générales sur la nature de la cible:

1. Génération de code intermédiaire
2. Optimisation de code
3. Génération effective du code final

Pour définir le code intermédiaire, on suppose une machine idéale représentative de la classe visée. La traduction de l'arbre abstrait est simple et « locale » (on peut traduire chaque nœud en partant de la traduction de ses fils).

Code intermédiaire

```
temp_1 := Ent-Vers-Fl (60)
temp_2 := id_3 * temp_1
temp_3 := id_2 + temp_2
id_1 := temp_3
```

L'optimisation utilise des techniques très générales qui elles aussi sont souvent « locale » (on balaye le code intermédiaire avec une fenêtre relativement étroite).

Code amélioré

```
temp_2 := id_3 * 60.0
id_1 := id_2 + temp_2
```

Le code objet final est évidemment étroitement dépendant de la cible.

Code objet

```
MOVF     id_3, R2
MULF     #60.0, R2
MOVF     id_2, R1
ADDF     R2, R1
MOVF     R1, id_1
```

Les fabricants de compilateur ont évidemment intérêt à proposer plusieurs compilateurs du même langage produisant du code pour des cibles différentes. La structuration que nous venons d'esquisser permet de réutiliser au mieux l'investissement initial: la phase ultime est la seule à dépendre de la machine cible.

Remarquons enfin que le langage objet peut être un langage qui demande encore à être traduit. Ainsi de nombreux compilateurs génèrent un code objet en C qui doit être ultérieurement confié à un compilateur C.

1.2.5 Schéma global

La structure d'un compilateur peut donc être résumée par le schéma suivant:

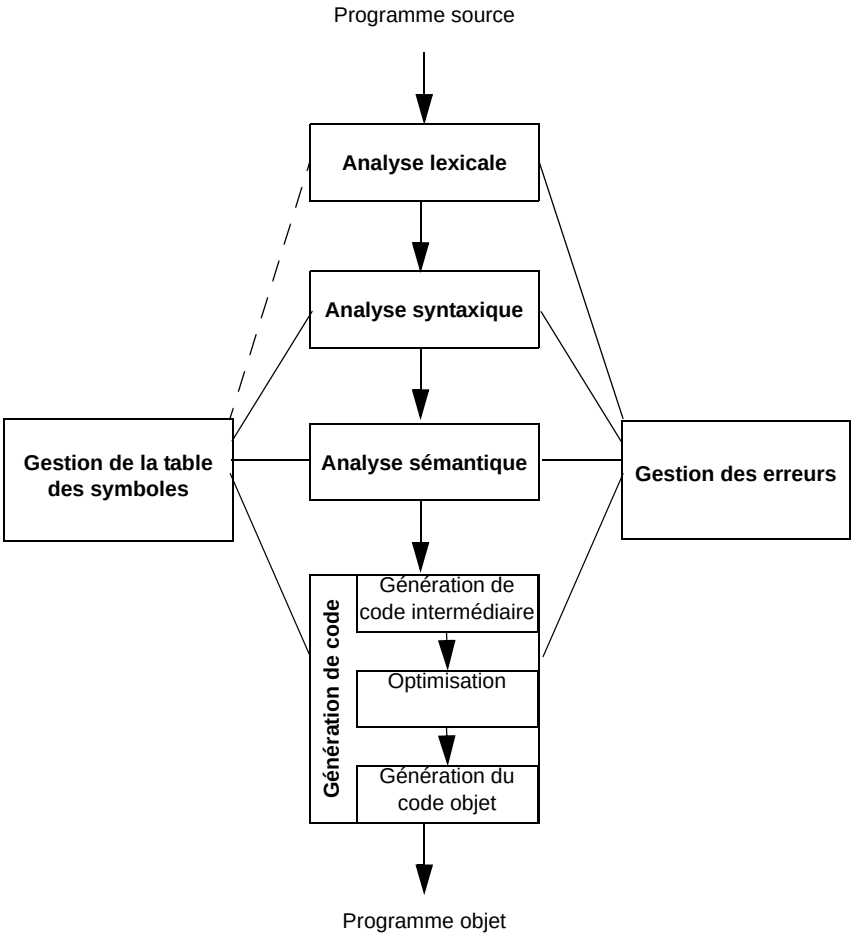


Figure 1.3 Structure d'un compilateur

1.2.6 Outils pour la construction de compilateurs

On dispose d'outils d'aide à la réalisation de compilateurs. Ces outils sont en général adaptés à l'une des phases répertoriées plus haut et leur disponibilité est variable.

- Des générateurs d'analyseurs lexicaux ou syntaxiques sont d'un usage courant depuis près d'un quart de siècle. Certains d'entre eux appartiennent au domaine public.
- Pour ce qui concerne l'analyse sémantique, on dispose, en laboratoire, de moteurs de traduction dirigés par la syntaxe: il s'agit d'associer des actions de traduction aux nœuds de l'arbre syntaxique.

- Pour assister la génération de code les fabricants de compilateurs disposent de générateurs automatiques de code, d'analyseurs de flots de données…

1.3 Autres membres de la famille

D'autres logiciels sont souvent associés aux compilateurs et interviennent dans le processus de traduction de programmes. Les plus fréquemment cités sont les pré-processeurs (angl. *preprocessors*), les assembleurs (angl. *assemblers*), les éditeurs de liens (angl. *linkers*) et les chargeurs (angl. *loaders*).

- Un pré-processeur est un programme qui effectue des modifications textuelles (souvent relativement simples) sur le texte source; en général le résultat du travail d'un pré-processeur doit être ultérieurement compilé.
- Un assembleur est un compilateur pour un type de langage très proche du langage machine.
- Un éditeur de liens a pour but de réunir plusieurs fragments de programme qui ont été compilés indépendamment les uns des autres.
- Un chargeur a pour fonction d'implanter le code exécutable dans la mémoire centrale et de lancer son exécution.

Édition de liens et chargement sont deux étapes incontournables (mais souvent « cachées ») dans le processus conduisant à l'exécution d'un programme.

1.4 Théorie des langages

Le français est un langage, Pascal également. Le but de la théorie des langages et de donner un modèle de ce qu'est un langage. Pour ce faire on utilise l'outil de base des scientifiques: les mathématiques. Cependant, le modèle en question est un modèle qualitatif et non quantitatif: son principal intérêt (à nos yeux) est de contribuer à améliorer l'activité de programmation. Enfin, il faut admettre d'emblée le caractère réducteur, donc incomplet, de toute modélisation.

Nous demandons au modèle une aide pour résoudre deux problèmes étroitement liés:

- être capable de décrire un langage;
- fabriquer une machine capable de reconnaître les textes qui appartiennent à un langage donné.

Ces deux problèmes, *description* et *reconnaissance*, recèlent une difficulté intrinsèque: il faut donner une description finie d'un objet en général infini: il y a en effet une infinité de textes français, une infinité de programmes Pascal.

1.4.1 Vocabulaire et mots

La réalité est multiple:

- Les mots du français sont composés en mettant les uns derrière les autres des caractères d'imprimerie.
- Une phrase française est composée en mettant les uns derrière les autres des mots du français.
- Un programme Pascal est obtenu en mettant les uns derrière les autres des mots-clés du langage Pascal.

Mettre les uns derrière les autres des éléments semble être une fonctionnalité de base d'un langage. D'où l'idée de retenir, dans le modèle, une généralisation.

Définition 1.4 Un *vocabulaire* V (ou alphabet, ou lexique) est un ensemble *fini* de symboles (ou caractères).

Définition 1.5 Un *mot* (ou phrase) sur un vocabulaire V est une séquence finie d'éléments de V.

Exemples

Vocabulaire	Mots
{a, b}	aaaa, aabb
{un, le, beau, féroce, chat, rat, mange, aime, qui}	un beau chat mange un le un le aime aime
{if, then, else, begin, end, :=, ;, (,), A, B, C, 1, 2}	if A then B := 1 else C := 2 if if if A begin

Il y a un mot particulier, la séquence vide; on l'appelle le *mot vide* et nous le notons ε.

Étant donné un mot m, on note |m| sa longueur, c'est-à-dire le nombre de symboles qui le composent.

Remarque
Il ne faut pas confondre la terminologie du modèle (vocabulaire, mot) et la terminologie propre au langage étudié (mot, phrase, programme…). Ainsi, si on s'intéresse au langage français, *Les misérables* de Victor HUGO pourra être considéré comme:

- un mot sur le vocabulaire de l'imprimerie;
- un mot sur le vocabulaire du lexique français.

Etant donné un vocabulaire V, on note V* l'ensemble des mots sur V. On peut munir V* d'une loi de composition interne: en mettant deux mots à la suite l'un de l'autre on obtient un nouveau mot.

Définition 1.6 L'opération de *concaténation* ■ est définie par:

```
■ :  V* x V*     →    V*
     a, b        →    le mot commençant par les symboles de a
suivis des
                      symboles de b.
```

Notations

- On notera a ■ b par ab. On remarque que |a ■ b| = |a| + |b|.
- On note a^n le mot composé de n occurrences de a. La notation a^0 désigne donc le mot vide.

L'opération de concaténation est associative et admet un élément neutre, le mot vide, ce qui confère à V* une structure de monoïde: on appelle V* le *monoïde libre* engendré par V. Le qualificatif libre tient au fait que les symboles sont considérés en dehors de toute signification.

Remarque
Sur V = {chat, te, chatte}, il est clair que chat ■ te est différent de chatte.

1.4.2 Langages

Pour un problème donné (le français, Pascal…) les mots de V* ne sont pas tous intéressants (cf. les exemples de mots donnés plus haut). D'où la définition de ce qu'est un langage.

Définition 1.7 Un langage L sur V est une partie de V*.

Un langage est donc, tout simplement, *un ensemble de mots*.

Exemples

Vocabulaire	Langage
{a, b}	L1 = {$a^n b^n$ | n >= 0} = { ε, ab, aabb, aaabbb, …}
{un, le, beau, féroce, chat, rat, mange, aime, qui}	L2 = ensemble des phrases françaises qu'on peut construire sur le lexique défini par le vocabulaire considéré
{if, then, else, begin, end, :=, ;, (,), A, B, C, 1, 2}	L3 = ensemble des conditionnelles Pascal limitées au lexique défini par le vocabulaire considéré

Pour aller plus loin, il nous faut un moyen de décrire sans ambiguïté ce qu'est un langage, les descriptions informelles se prêtant très mal à des traitements in-

formatiques. L'un des buts de ce cours est d'introduire formellement les notions duales de grammaire (*description*) et d'automate (*reconnaissance*).

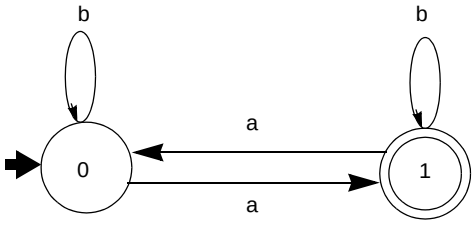
CHAPITRE 2

Automates à nombre fini d'états et langages rationnels

2.1 Automates déterministes à nombre fini d'états

2.1.1 Présentation informelle

Le dessin qui suit représente un automate déterministe à nombre fini d'états qui reconnaît le langage des mots sur {a, b}* qui comptent un nombre impair de a.



À tout instant, cette machine est dans un des états (0 ou 1) et possède une tête de lecture qui repère un des caractères du mot à analyser.

Au départ, la machine est dans l'état initial (0), la tête de lecture étant positionnée sur le premier caractère du mot.

La machine effectue une suite de transitions, gouvernées par les flèches du dessin, l'étiquette d'une flèche devant correspondre au caractère sous la tête de lecture pour permettre une transition. À chaque transition la tête est déplacée d'un caractère vers la droite.

Lors de l'arrêt, la machine est dans l'une des trois configurations suivantes:

- Le mot n'est pas entièrement lu mais aucune transition n'est possible: le mot est rejeté.
- Le mot a été lu, mais l'état dans lequel se trouve la machine n'est pas un état final: le mot est rejeté.
- Le mot a été lu, et l'état dans lequel se trouve la machine est un état final, marqué par un double cercle: le mot **est** accepté (on dit aussi *reconnu*)..

2.1.2 Définition

Définition 2.1 Un automate déterministe M à nombre fini d'états est un quintuplet $M = (Q, V, \delta, q_0, F)$ où

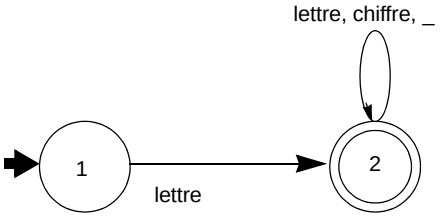
- Q est ensemble fini d'*états*
- V est un vocabulaire
- δ une *fonction de transition* de Q x V dans Q
- q_0, l'*état initial* est un élément de Q
- F, l'ensemble des *états finaux*, est inclus dans Q

La fonction de transition de l'automate qui précède est la suivante:

δ	**a**	**b**
0	1	0
1	0	1

2.1.3 Automates complets

Un automate capable de reconnaitre le langage des identificateurs Eiffel est le suivant.



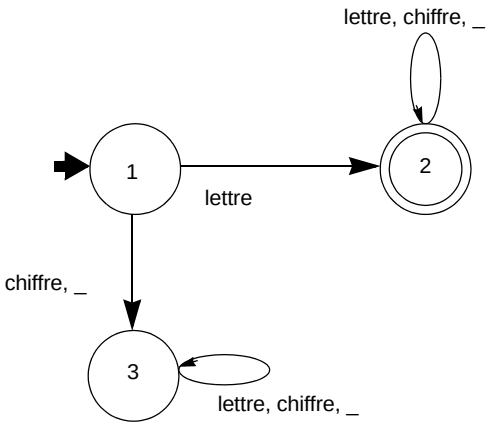
Sa fonction de transition est une fonction partielle.

δ	**lettre**	**chiffre, _**
1	2	indéfini
2	2	2

Toute fonction de transition partielle peut être complétée en ajoutant un état supplémentaire, traditionnellement appelé *état puits*:

δ	**lettre**	**chiffre, _**
1	2	3
2	2	2
3	3	3

On obtient alors un automate, dit *automate complet*, équivalent à l'automate de départ.



La capacité de compléter ou non un automate a un intérêt pratique indéniable: selon le cas, on peut souhaiter que la machine s'arrête dès qu'un élément de la bande d'entrée n'est pas admissible ou, au contraire, on peut vouloir que la bande d'entrée soit systématiquement consommée.

Il est intéressant, dans certains cas, de normaliser la présentation des automates: il s'agit de les compléter et d'enlever les états inaccessibles.

2.1.4 Langage accepté par un automate

Définition 2.2 Une *configuration* d'un automate est un couple (q, w) où q est l'état courant et w le mot qui reste à analyser.

Définition 2.3 Une *transition* d'un automate relie deux configurations successives:
$(q, aw) \rightarrow_M (p, w)$ si et seulement si $\delta(q, a) = p$

Définition 2.4 Une séquence de transitions relie deux configurations: $(q, w) \rightarrow^*_M (p, v)$ si et seulement si

- soit q = p et w = v

- soit $(q, w) \rightarrow_M (r, u)$ et $(r, u) \rightarrow^*_M (p, v)$

Définition 2.5 Le langage L(M) accepté par un automate M est défini par:
$L(M) = \{w \mid (q_0, w) \rightarrow^*_M (f, \varepsilon) \text{ avec } f \in F\}$

2.1.5 Notations et variantes terminologiques

Lorsque $(q, w) \rightarrow^*_M (p, \varepsilon)$ il existe une suite d'applications de la fonction de transition qui permet, à partir de q, de « lire » entièrement le mot w et « d'arriver » dans l'état p. Il est commode de disposer de la fonction $\hat{\delta}$ qu'on peut définir de deux manières équivalentes:

- $\hat{\delta}(q, w) = p$ si et seulement si $(q, w) \rightarrow^*_M (p, \varepsilon)$
- $$\begin{cases} \hat{\delta}(q, \varepsilon) = q \\ \hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w) \end{cases}$$

Pour exprimer le fait qu'il existe une « flèche » étiquetée par a reliant q à p il suffit d'écrire $\delta(q, a) = p$. Pour désigner une telle flèche, nous utilisons également la notation $(q, a, p) \in \delta$.

2.2 Programmation par automates

2.2.1 Un problème de programmation

On se pose le problème de programmation suivant: sur un fichier d'entrée on s'attend à trouver des entiers séparés par une des marques « h », « mn », « s ». Les caractères espace (« _ ») et fin de ligne (« ↵ ») jouent le rôle de séparateurs et le caractère « . » de marqueur de fin.

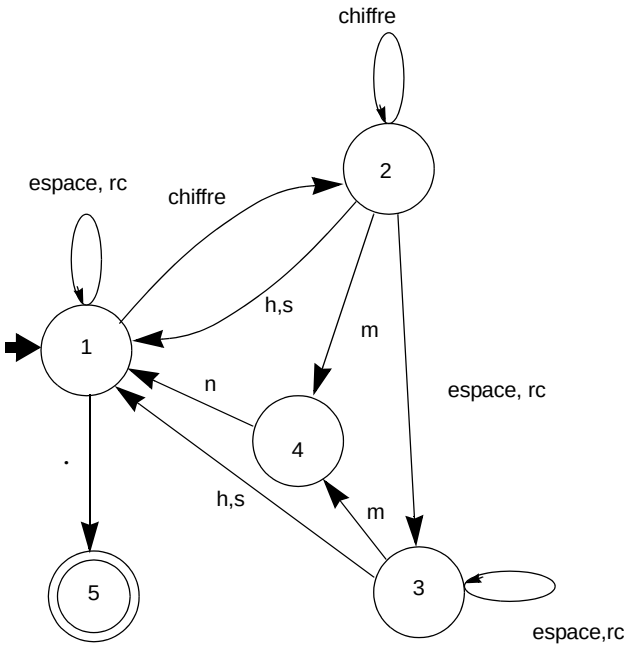
Exemples

```
___12h1s__12__mn_↵
1s1h2h__3____s↵
↵
.↵
```

On peut interpréter les informations de ce fichier comme une durée dont on veut calculer la valeur totale en secondes.

2.2.2 Reconnaissance

On ne s'intéresse, dans un premier temps, qu'à déterminer si les informations du fichiers correspondent bien à ce qu'on attend. On peut modéliser le problème en donnant un automate déterministe qui décrit les fichiers corrects.



On remarque que la donnée d'un tel automate constitue également une spécification précise du langage des données et que le rédacteur de ces spécifications a dû lever les imprécisions de l'énoncé informel donné plus haut. Ainsi:

- on ne peut laisser d'espace entre le « `m` » et le « `n` » de la marque « `mn` »;
- on peut laisser des espaces entre un nombre et la marque qui le suit;
- un nombre peut commencer par des zéros non significatifs;
- on ne peut insérer des espaces dans un nombre;
- une durée vide est tolérée…

Le lecteur s'exercera avec profit à faire d'autres choix et à donner des automates adéquats.

2.2.3 Association d'actions aux transitions

Pour résoudre un problème de programmation dont la spécification comporte un automate déterministe à nombre fini d'états, on peut adopter une démarche de programmation peu systématique:

- on utilise un programme général qui mime le comportement d'un automate; ce programme gère les entrées/sorties (tête de lecture) et l'état courant.

- à chaque transition de l'automate, on associe une action qui est une séquence de code exécutée lorsque la transition concernée est effectuée.

Les actions utilisent des structures de données qui doivent évidemment être déclarées et éventuellement initialisées. On associe également une séquence de code à l'arrêt de l'automate.

Dans ce cadre, une solution au problème qui nous intéresse est la suivante:

Déclarations	`valent, cumul:ENTIER;` `erreur:BOOLÉEN ;`

Initialisations	`valent := 0 ;`
	`cumul := 0;`
	`erreur := false ;`

Action finale	**si** ¬ erreur **alors** affi- cher(cumul) **fsi**

Numéro d'action	Transitions concernées	Calculs
0	erreurs	`erreur := true ;`
1	1, chiffre, 2	`valent := Tete - charcode('0') ;`
2	2, chiffre, 2	`valent := valent * 10 + Tete - charcode('0') ;`
3	2, h, 1 3, h, 1	`cumul := cumul + 3600 * valent ;`
4	2, s, 1 3, s, 1	`cumul := cumul + valent ;`
5	4, n, 1	`cumul := cumul + 60 * valent ;`
6	autres	

2.3 Opérations sur les langages

On peut songer à définir un langage en composant des langages déjà définis. Trois opérations de composition, *l'union*, le *produit* et la *fermeture*, se révèlent particulièrement utiles.

2.3.1 Définitions

L'union et le produit correspondent à des opérations assez naturelles.

Définition 2.6 L'*union* `L1 + L2` de deux langages est l'union ensembliste habituelle. Elle est donc définie par:

- $L1 + L2 = \{ m \mid m \in L1 \vee m \in L2 \}$

Définition 2.7 Le *produit* L1L2 de deux langages est défini par:

- $L1L2 = \{m1m2 \mid m1 \in L1 \wedge m2 \in L2 \}$

Définition 2.8 La *fermeture* L* d'un langage L correspond intuitivement à une itération. Elle comprend:

- L_0: le mot vide (pas 0)
- L_1: tous les mots obtenus en concaténant un mot de L_0 et un mot de L (pas 1)
- L_2: tous les mots obtenus en concaténant un mot de L_1 et un mot de L (pas 2)
- ...

Le langage L_i défini au pas i est donc $L_{i-1}L$ et L* est l'union infinie des L_i.

2.3.2 Exemples

- L1 = {a} + {b}
- L2 = L1* est le monoïde libre.
- L3 = {a}L2{a} est l'ensemble des mots qui commencent et qui finissent par a.
- L4 = {b}L2{b} est l'ensemble des mots qui commencent et qui finissent par b.
- L5 = L3 + L4 est l'ensemble des mots qui soit commencent et finissent par a, soit commencent et finissent par b.
- L6 = L2L5L2 est l'ensemble des mots qui comportent au moins soit 2a soit 2b.

2.3.2.1 Expressions rationnelles

Définition 2.9 Les *expressions rationnelles* (ou encore *régulières*) sont définies récursivement par les règles suivantes:

- $\varnothing$ est une expression rationnelle qui représente le langage $\varnothing$.
- Si $a \in V$, a est une expression rationnelle qui représente le langage {a}.
- ε est une expression rationnelle qui représente le langage $\{\varepsilon\}$.
- Si a et b sont des expressions rationnelles qui représentent les langages A et B, (a + b) est une expression rationnelle qui représente le langage A + B.
- Si a et b sont des expressions rationnelles qui représentent les langages A et B, (a b) est une expression rationnelle qui représente le langage AB.
- Si a est une expression rationnelle qui représente le langage A, (a)* est une expression rationnelle qui représente le langage A*.

Pour alléger l'écriture on adopte des conventions de parenthésage, les opérateurs étant munis d'une priorité. Dans l'ordre décroissant : *, produit, +.

Exemples

- (a + b)* représente le monoïde libre.
- (a + b)*aa(a + b)* représente l'ensemble des mots contenant au moins 2a consécutifs
- si lettres = A + ... + Z + a + ... + z et chiffres = 0 + ... + 9 alors lettres(lettres + chiffres + _)* représente le langage des identificateurs.

On peut ainsi caractériser L6 par (a + b)*(a(a +b)*a + b(a + b)*b)(a + b)*.

2.3.3 Classe des langages rationnels

Définition 2.10 Un langage rationnel est un langage construit à partir de langages finis en utilisant un nombre fini de fois des opérations d'union, de produit et de fermeture de langages.

Un langage fini est un langage comportant un nombre fini de mots. On remarque que l'opération de fermeture est capable de produire un langage infini à partir d'un langage fini.

Comme un langage fini peut être décrit à partir des éléments du vocabulaire en utilisant un nombre fini de fois l'union et le produit, on constate que, selon notre définition, la classe des langages rationnels est celle des langages descriptibles par une expression régulière. On désigne d'ordinaire par Rat(V) la classe des langages rationnels sur le vocabulaire V.

2.3.4 Vers des automates indéterministes

Il est facile de construire un automate à nombre fini d'états capable de reconnaître un langage fini donné.

S'il était possible de définir l'union, le produit et la fermeture sur les automates à nombre fini d'états on aurait une inclusion de la classe Rat(V) dans la classe Rec(V) des langages reconnaissables par un automate à nombre fini d'états. La capacité de passer d'un formalisme de description à un formalisme de reconnaissance est évidemment très profitable d'un point de vue pratique.

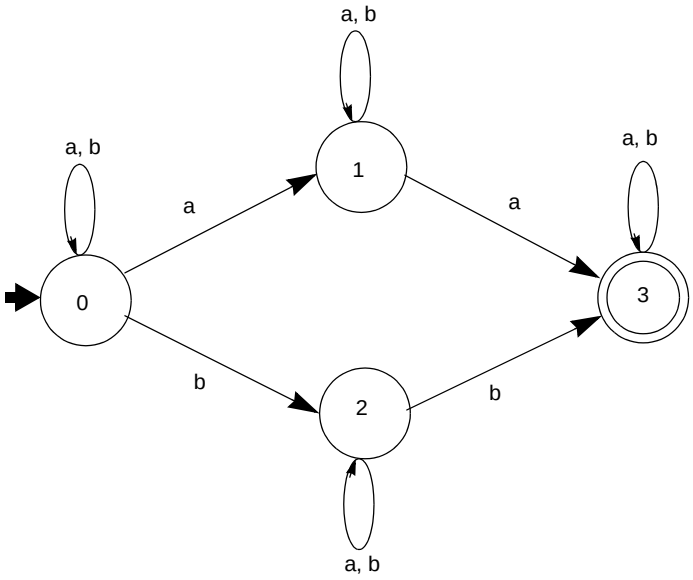
Il se trouve que les combinaisons simples d'automates qui viennent à l'esprit ont un inconvénient majeur: l'objet obtenu n'est pas, dans le cas général, un automate déterministe à nombre fini d'états. Pour tourner cette difficulté on peut chercher à étendre la notion d'automate et définir ce qu'on appelle un automate indéterministe à nombre fini d'états.

2.4 Automates indéterministes

2.4.1 Présentation informelle

Considérons le langage L = (a + b)*(a(a +b)*a + b(a + b)*b)(a + b)*

Le dessin qui suit est simple et parlant.



C'est un automate *indéterministe* (lors de certaines transitions, la machine a le choix). Un tel automate accepte un mot si, parmi tous les choix possibles, il y a une séquence de transitions qui aboutit à un état final.

Il n'est pas commode d'implanter pratiquement un automate indéterministe. Leur intérêt majeur réside dans le fait qu'ils sont faciles à concevoir et qu'on dispose d'un algorithme pour les déterminiser.

2.4.2 Définitions

Définition 2.11 Un automate M à nombre fini d'états est un quintuplet M = (Q, V, δ, q_0, F) où

- Q est ensemble fini d'*états*
- V est un vocabulaire
- δ une *fonction de transition* de Q x V dans Q (déterministe) ou $\mathcal{P}$(Q) (indéterministe)
- q_0, l'*état initial* est un élément de Q
- F, l'ensemble des *états finaux*, est inclus dans Q

La fonction de transition de l'automate qui précède est:

δ	**a**	**b**
0	{0,1}	{0, 2}
1	{1, 3}	{1}
2	{2}	{2, 3}
3	{3}	{3}

On généralise sans peine les définitions et notations données dans le cas déterministe en modifiant la définition de la relation de transition entre deux configurations.

Définition 2.12 Une *transition* d'un automate indéterministe relie deux configurations:
$(q, aw) \; * \; (p, w)$ si et seulement si $\delta(q, a) \in p$

2.4.3 Déterminisation

Propriété 2.1 Soit $M = (Q, V, \delta, q_0, F)$ un automate indéterministe. Il existe au moins un automate déterministe $M' = (Q', V, \delta', q'_0, F')$ tel que $L(M) = L(M')$.

On peut construire M' de la manière suivante:

- $Q' = \mathcal{P}(Q)$
- $q'_0 = \{q_0\}$
- F' contient tous les sous-ensembles S de Q tels que $S \cap F \neq \varnothing$
- Pour tout $S \subseteq Q$, $\delta'(S, a) = S'$, où $S' = \{p \mid p \in \delta(q, a) \text{ et } q \in S\}$

Sur l'exemple qui précède

δ	**a**	**b**
0	{0,1}	{0, 2}
1	{1, 3}	{1}
2	{2}	{2, 3}
3	{3}	{3}

on obtient, en se limitant aux états accessibles

δ '	**a**	**b**
{0}	{0,1}	{0, 2}

δ '	**a**	**b**
{0, 1}	{0, 1, 3}	{0, 1, 2}
{0, 2}	{0, 1, 2}	{0, 2, 3}
{0, 1, 3}	{0, 1, 3}	{0, 1, 2, 3}
{0, 1, 2}	{0, 1, 2, 3}	{0, 1, 2, 3}
{0, 2, 3}	{0, 1, 2, 3}	{0, 2, 3}
{0, 1, 2, 3}	{0, 1, 2, 3}	{0, 1, 2, 3}

Les états finaux sont {0, 1, 3}, {0, 2, 3} et {0, 1, 2, 3}.

2.5 Minimisation

2.5.1 Automate minimum

Considérons le langage L = (a + b)*aa(a + b)*. On peut facilement donner, à partir de l'expression régulière, un automate indéterministe M.

M

d	**a**	**b**
0	{0,1}	{0}
1	{2}	ý
2	{2}	{2}

En appliquant l'algorithme précédent, on obtient l'automate déterministe M1 qui suit.

M1

δ_1	**a**	**b**
{0}	{0,1}	{0}
{0, 1}	{0, 1, 2}	{0}
{0, 1, 2}	{0, 1, 2}	{0, 2}
{0, 2}	{0, 1, 2}	{0, 2}

On constate facilement que l'automate M2 admet le même langage.

M2

δ_2	a	b
0	1	0
1	2	0
2	2	2

Cet exemple nous permet de faire deux constatations importantes:

- Parmi tous les automates déterministes qui décrivent le même langage il y en a qui ont un nombre minimum d'états. Il n'est pas difficile de se convaincre que les automates complets minimaux sont identiques si on fait abstraction du nom des états.
- L'algorithme de déterminisation ne produit pas obligatoirement l'automate minimum.

Il est possible de disposer d'un algorithme permettant de calculer un automate minimum équivalent à un automate déterministe donné.

2.5.2 Algorithme de minimisation

Définition 2.13 Étant donné un automate déterministe M on dit que deux états p et q sont *i-équivalents* si et seulement si, pour tout m de longueur inférieure ou égale à i

- $\hat{\delta}(p, m) \in F \Leftrightarrow \hat{\delta}(q, m) \in F$

En d'autres termes, à partir de p ou de q « on accepte et on rejette exactement les mêmes mots de longueur inférieure ou égale à i ». On note R_i la relation de i-équivalence.

On a une caractérisation simple de la 0-équivalence et on peut définir facilement la i-équivalence à partir de la (i -1)-équivalence:

- $pR_0q \Leftrightarrow (p \in F \wedge q \in F) \vee (p \notin F \wedge q \notin F)$
- $pR_iq \Leftrightarrow pR_{i-1}q \wedge \forall a \in V,\ \delta(p, a)R_{i-1}\delta(q, a)$

On constate que, si il existe un k tel que $R_{k+1} = R_k$ alors, pour tous les n plus grands que k, $R_n = R_k$. De plus, comme le nombre de classes d'équivalence est borné par |Q|, un tel k existe forcément.

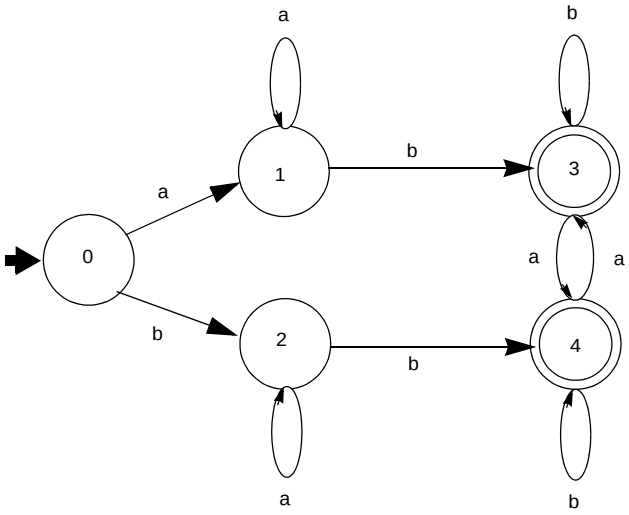
De ce qui précède, on peut déduire un algorithme de minimisation.

Algorithme de minimisation

```
1. Enlever les états inaccessibles et compléter l'automate de
départ.
```

2. Poser R_0 = {F, Q - F}
3. Itérer le calcul de Ri jusqu'à ce que $R_i = R_{i-1}$
4. Soit R le résultat final. On pose $M_R = (V, Q/R, D_R, Q_{0R}, F/R)$[1]

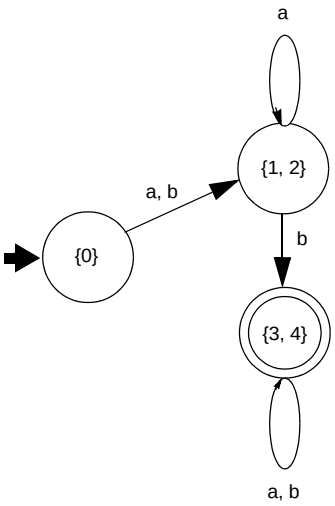
Exemple



1. L'automate est complet et ne comporte pas d'états inaccessibles
2. R0 contient les deux classes F et Q-F ; posons R = ({3, 4} {0, 1, 2})
3.
    R1.
        - 0 se sépare de 1 car $\delta(0, b) \in \{0, 1, 2\}$ alors que $\delta(1, b) \in \{3, 4\}$
        - 1 ne se sépare pas de 2 car
            $\delta(1, a) \in \{0, 1, 2\}$ et $\delta(2, a) \in \{0, 1, 2\}$
            $\delta(1, b) \in \{3, 4\}$ et $\delta(2, b) \in \{3, 4\}$
        - 3 ne se sépare pas de 4
        D'où R = ({3, 4} {0} {1, 2})
    R2.
        aucune séparation

1. Les ensembles Q/R et F/R sont les ensembles des classes d'équivalence, pour R, de Q et F. Q_{0R} est l'élément de Q/R contenant q_0. $D_R(Q, a)$ est la classe contenant les images $\delta(q, a)$ des éléments q de Q.

4. Le résultat est l'automate M_R qui suit

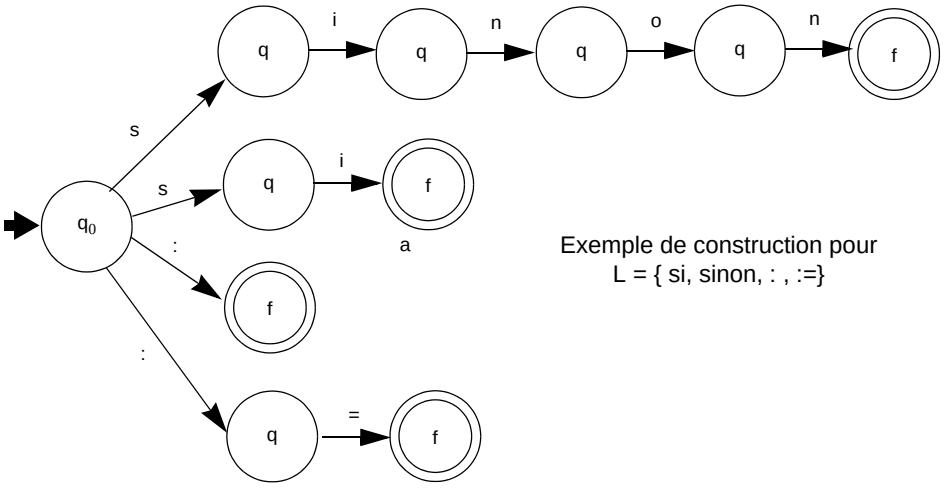


2.6 Classe des langages rationnels

La classe des langages rationnels et la classe des langages reconnaissables sont identiques.

2.6.1 Inclusion de la classe des rationnels dans la classe des reconnaissables

1. On peut facilement construire un automate à nombre fini d'états reconnaissant un langage fini donné.



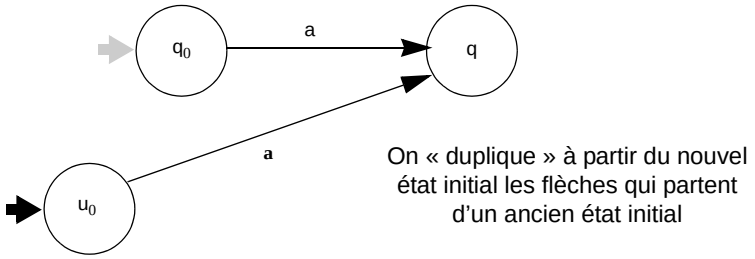
Exemple de construction pour
L = { si, sinon, : , :=}

2. Étant donné 2 automates M et M', un automate U reconnaissant L(M) ∪ L(M') peut-être défini par:

$$U = (Q \cup Q' \cup \{u_0\}, V, u_0, d, F \cup F' \cup N)$$

avec $d = \delta \cup \delta' \cup \{(u_0, a, q) \mid (q_0, a, q) \in \delta \vee (q'_0, a, q) \in \delta'\}$

et $N = \varnothing$ **si** $q_0 \notin F \wedge q'_0 \notin F'$, $\{u0\}$ **sinon**

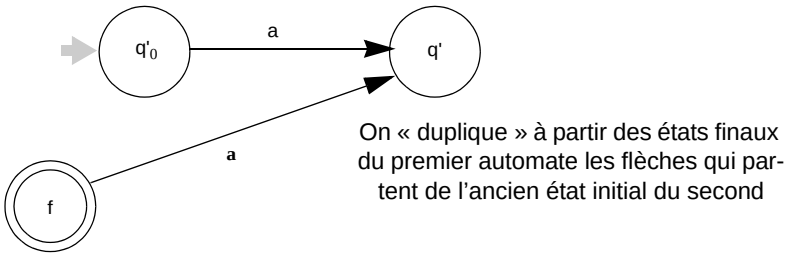


3. Étant donné 2 automates M et M', un automate P reconnaissant L(M)L(M') peut-être défini par:

$P = (Q \cup Q', V, q_0, d, F' \cup N)$

avec $d = \delta \cup \delta' \cup \{(f, a, q') \mid f \in F \vee (q'_0, a, q') \in \delta'\}$

et $N = \varnothing$ **si** $q'_0 \notin F'$, F **sinon**

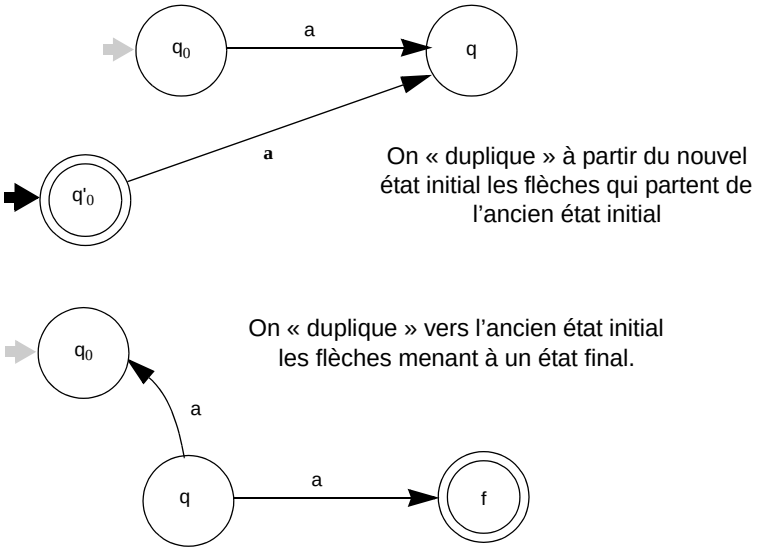


4. Étant donné un automate M, un automate B reconnaissant L(M)* peut-être défini par:

$B = (Q, V, q'_0, d, F \cup \{q'_0\})$

avec

$d = \delta \cup \{(q'_0, a, q) \mid (q_0, a, q) \in \delta\} \cup \{(q, a, q_0) \mid \exists f \in F, (q, a, f) \in \delta\}$



Des quatre propriétés qui précèdent on peut déduire que Rat(V) ⊆ Rec(V).

2.6.2 Inclusion de la classe des reconnaissables dans la classe des rationnels

Considérons un automate $M = (\{q_1, ..., q_n\}, V, q_1, \delta, F)$ et posons

$R^k_{i,j}$ = {x | x est la trace d'un chemin allant de q_i à q_j dont tous les sommets intermédiaires q_p vérifient $p \leq k$}

On remarque que:

1. $R^0_{i,j}$ correspond aux flèches directes entre i et j
2. $R^k_{i,j} = R^{k-1}_{i,j} \cup R^{k-1}_{i,k} (R^{k-1}_{k,k})^* R^{k-1}_{k,j}$
3. $L(M) = \bigcup_{q_f \in F} R^n_{1,f}$

Le langage accepté par un automate est descriptible par une expression rationnelle, donc Rec(V) ⊆ Rat(V).

2.7 Quelques propriétés des langages rationnels

2.7.1 Lemme dit de l'étoile

Propriété 2.2 Soit L ∈ Rec(V), alors il existe un entier n tel que pour tout mot de L de longueur supérieure ou égale à n on peut trouver 3 mots x, y et z tels que:

- m = xyz

- $y \neq \varepsilon$
- $\forall k, xy^kz \in L$

En effet, dès que la trace d'un mot reconnaissable dépasse le nombre d'états de l'automate, elle passe au moins 2 fois par le même état et lorsque le sous-mot ainsi délimité est répété un nombre quelconque de fois, on obtient toujours, par construction, la trace d'un mot reconnaissable.

Cette propriété permet surtout de montrer que des langages ne sont pas rationnels.

Exemple
Considérons $L = \{a^ib^i \mid i \geq 0\}$. Si L est rationnel, d'après le lemme de l'étoile il existe un certain rang N à partir duquel un sous-mot de a^Nb^N doit pouvoir être répété. Comme c'est impossible on en déduit que L n'est pas rationnel.

2.7.2 Autres propriétés

Propriété 2.3 On sait décider de l'équivalence de 2 langages rationnels.
Il suffit en effet de calculer les 2 automates déterministes minimaux et de les comparer.

Propriété 2.4 Le complémentaire d'un langages rationnel est rationnel.
On montre facilement que l'automate obtenu en échangeant états finaux et non finaux d'un automate déterministe minimum M on obtient un automate reconnaissant le complémentaire de $L(M)$.

2.8 Principales applications

2.8.1 Analyse lexicale

Les éléments lexicaux d'un langage de programmation (mots clés, symboles de ponctuation, identificateurs, constantes) forment un langage d'ordinaire rationnel. Il est commode de décrire ces éléments lexicaux en donnant des expressions rationnelles et de structurer cette description en catégories lexicales.

Il est alors possible de calculer automatiquement un analyseur lexical. On procède de la manière suivante:

1. Calcul d'un automate non-déterministe équivalent à l'expression rationnelle.
2. Déterminisation et minimisation.
3. Production des tables de transition de l'automate, auxquelles on associe des actions simples: stockage de l'élément lexical, production de sa catégorie.

2.8.2 Édition de texte et recherche de motifs

De nombreux algorithmes utilisés par les outils de traitement de texte calculent ou utilisent des automates à nombre fini d'états.

2.8.3 Saisie de données

De nombreuses applications comportent des phases de saisies de données dont le langage est rationnel. L'utilisation de la programmation par automate présente quatre avantages principaux:

1. On obtient une spécification précise du format des données.
2. On n'est pas tenté d'apporter des restrictions arbitraires « justifiées » par l'idée souvent fausse d'obtenir un code plus simple.
3. On réduit l'effort de programmation à l'essentiel, tout en réduisant le risque d'erreur.
4. Le logiciel obtenu est plus évolutif.

2.8.4 Applications parallèles ou réactives

Citons, pour terminer, le fait que la notion d'automate à nombre fini d'états est un des modèles particulièrement utiles pour concevoir, modéliser, tester… des applications réactives (temps réel entre autres) ou parallèles (protocoles…).

CHAPITRE 3

Grammaires et langages

3.1 Définition

3.1.1 Présentation informelle

Les grammaires formelles que nous présentons dans ce chapitre ont leur origine dans les travaux de Noam CHOMSKY; il s'agit d'un formalisme assez général permettant de décrire un langage. Ce formalisme est proche de la notion usuelle de grammaire et il repose sur l'utilisation d'un mécanisme génératif capable de produire tous les mots d'un langage donné.

Reprenons trois exemples de langages utilisés au chapitre 1.

Vocabulaire	Langage
{a, b}	L1 = {$a^n b^n$ | n >= 0} = { ε, ab, aabb, aaabbb, …}
{un, le, beau, féroce, chat, rat, mange, aime, qui}	L2 = ensemble des phrases françaises qu'on peut construire sur le lexique défini par le vocabulaire considéré
{if, then, else, begin, end, :=, ;, (,), A, B, C, 1, 2}	L3 = ensemble des conditionnelles Pascal limitées au lexique défini par le vocabulaire considéré

Pour certains mots de ces langages, nous essayons de donner une « explication » de leur appartenance au langage.

Mot	Explication
aabb	un mot de L1 peut être un a, suivi d'un mot de L1, suivi d'un b.
le chat mange le rat	une phrase française peut être un groupe nominal suivi d'un groupe verbal.
if A **then** B := 1	une conditionnelle Eiffel peut être **if** suivi d'une expression, suivi de **then**, suivi d'une instruction, suivi de **end**.

La seconde explication est une des règles grammaire du français et on remarquera la similitude de structure avec les deux autres.

3.1.2 Vocabulaires

Pour exprimer les trois règles qui précèdent nous avons utilisé:

- des éléments du vocabulaire du langage, comme **if**, **then**…
- d'autres éléments (instruction, expression, groupe nominal…) qui ne sont pas quelconques et qui correspondent à la notion de catégorie grammaticale.

Ces deux ingrédients de base sont pris parmi deux vocabulaires, le vocabulaire *terminal* V et le vocabulaire *non-terminal* N. Ces deux vocabulaires sont choisis disjoints, pour éviter certaines ambiguïtés; en français courant, la phrase « verbe est un nom, nom est un nom, manger n'est pas un nom mais un verbe » demande d'ordinaire quelques instants de réflexion avant que sa signification ne soit clairement perçue. En utilisant une notation désormais courante en informatique, on pourrait écrire:

```
verbe est un <nom>, nom est un <nom>, manger n'est pas un <nom>
mais un <verbe>
```

La notation en question consiste à distinguer par des chevrons un élément du vocabulaire non-terminal (la catégorie grammaticale <verbe>) de l'élément du vocabulaire terminal correspondant (le mot verbe).

Deux systèmes de notation sont souvent utilisés.

- On réserve les majuscules pour N, les minuscules pour V: $A \in N$ et $a \in V$.
- On note les non-terminaux entre chevrons: <truc> $\in N$ et truc $\in V$.

A l'aide des deux vocabulaires on peut construire des mots sur $N \cup V$ (on les désigne d'ordinaire par une lettre grecque, $\alpha \in (N \cup V)^*$.

Exemples

- `<conditionnelle>`
- **if** `<expression>` **then** `<instruction>`
- `<relative>`
- **qui** `<groupe-verbal>`

3.1.3 Règles

Une *règle* $\alpha \rightarrow \beta$ met en relation une partie gauche non vide α et une partie droite, β qui sont deux éléments de $(N \cup V)^*$.

Exemples

- `<conditionnelle>` → **if** `<expression>` **then** `<instruction>`
- `<relative>` → `qui <groupe-verbal>`

3.1.4 Réécriture (dérivation)

Les règles permettent de réécrire des mots sur $N \cup V$.

Exemple

Règles utilisées	**Mots sur N ∪ V**
	`<conditionnelle> ; <condition-nelle>`
`<conditionnelle>` → **if** `<expression>` **then** `<ins-truction>`	**if** `<expression>` **then** `<instruction> ; <conditionnelle>`
`<expression>` → `<identi-ficateur>`	**if** `<identificateur>` **then** `<instruc-tion> ; <conditionnelle>`
`<identificateur>` → `<let-tre>`	**if** `<lettre>` **then** `<instruction> ; <conditionnelle>`
`<lettre>` → `A`	**if** `A` **then** `<instruction> ; <condi-tionnelle>`
…	…

Les réécritures peuvent aboutir à des mots de V^*. Si on sait de quoi partir, on sait donc produire des mots de V^*. Le point de départ est appelé *axiome*.

Exemple

N	V	Règles	Axiome
{S}	{a, b}	S → ε S → aSb	S

A partir de l'axiome, on peut réécrire, à l'aide des règles de G, les mots ε, ab, aabb…

- S peut être réécrit en ε
- S peut être réécrit en aSb, qui peut être réécrit en aaSbb, qui peut être réécrit en aabb.

On constate que tous les mots de V* ainsi produits sont de la forme $a^n b^n$.

3.1.5 Résumé

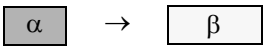
Définition 3.1 Une *grammaire* est un quadruplet G = (N, V, S, P) où

- V est un vocabulaire dit terminal qui est le vocabulaire du langage;
- N est un vocabulaire dit non-terminal, ($N \cap V = \varnothing$);
- S, appelé axiome, est un élément de N;
- P est un ensemble de règles de la forme $\alpha \rightarrow \beta$, $\alpha \neq \varepsilon$, où α et β appartiennent à $(N \cup V)^*$.

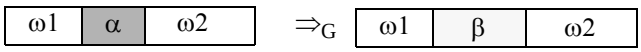
Définition 3.2 Etant donné une grammaire G et deux mots $\omega 1 \beta \omega 2$ et $\omega 1 \alpha \omega 2$ de $(N \cup V)^*$, on dit que $\omega 1 \beta \omega 2$ *dérive directement* de $\omega 1 \alpha \omega 2$ si et seulement si $\alpha \rightarrow \beta \in P$.

Lorsque γ dérive directement de δ, on utilisera la notation $\delta \Rightarrow_G \gamma$ et on dira aussi (autre terminologie) que δ se réécrit en γ.

Règle



Réécriture



Définition 3.3 Etant donné une grammaire G et deux mots α_0 et α_n de $(N \cup V)^*$, on dira que α_n *dérive* de α_0 (ou encore que α_0 se *réécrit* en α_n), ce qu'on notera $\alpha_0 \Rightarrow_G^* \alpha_n$, si et seulement si:

- soit $\alpha_0 = \alpha_n$
- soit $\alpha_0 \Rightarrow_G \alpha_1$ et $\alpha_1 \Rightarrow_G^* \alpha_n$, pour un certain α_1 de $(N \cup V)^*$

Définition 3.4 Etant donné une grammaire G, le *langage* L(G) *engendré* par G est défini par:

- $L(G) = \{ m \in V^* \mid S \Rightarrow_G^* m \}$

3.1.6 Exemples

3.1.6.1 V = {a, b}

- Langage des palindromes

```
S → a S a
S → b S b
S → a
S → b
S → ε
```

- Langage des mots comportant autant de a que de b

```
S → a B  A → a      B → b
S → b A  A → a S    B → b S
S → ε    A → b A A  B → a B B
```

Plutôt que de noter *in extenso* toutes les règles, on factorise d'ordinaire les parties gauches, en utilisant la notation qui suit.

```
S → a B | b A | ε
A → a | a S | b AA
B → b | b S | a B B
```

On remarque que plusieurs grammaires peuvent décrire le même langage.

```
S → a S b S | b S a S | ε
```

3.1.6.2 V = {a, b, c}

- Langage $a^n b^n c^n$, $n \geq 1$

```
S → a S B C   C B → B C      a B → a b
S → a B C                    b B → b b
                             b C → b c
                             c C → c c
```

3.1.6.3 Conditionnelles Pascal

Le vocabulaire du langage Pascal (norme de base) comporte 95 symboles:

- 26 lettres
- 10 chiffres
- 24 symboles de ponctuation: + - * / < > = . , ; : ( ) [ ] { } ' ¦ >= <= <> .. :=
- 35 mots-clés[1]: **and array begin case const div do downto else end file for from function goto if in label mod nil not or packed procedure program record repeat set then type to until var while with**

L'axiome du langage est le non-terminal <programme>. On a des règles comme:

```
<programme> → program <identificateur> ; <bloc> .
<programme> → program <identificateur> ( <liste-identificateurs> ) ; <bloc> .
```

1. On notera la différence entre mot-clé et identificateur réservé: integer et true ne sont pas des terminaux du langage. Ils peuvent être décrit par la grammaire.

On peut considérer que chaque non-terminal décrit un sous-langage. Pour le sous-langage des contionnelles on a:

```
<conditionnelle> → if <expression> then <énoncé>
<conditionnelle> → if <expression> then <énoncé> else <énoncé>

<énoncé> → <entier-sans-signe> : <instruction>
<énoncé> → <instruction>
<instruction> → <affectation> | <appel> | <bloc> | <condi-
tionnelle> | <cas> | <boucle> | <avec> | <branchement>
```

3.1.6.4 Bonchatbonrat

On considère le vocabulaire V[1] = {un, le, beau, féroce, chat, rat, mange, aime, qui}.

Les règles qui suivent définissent un sous-ensemble des phrases françaises.

```
<Phrase> → <Gn> <Gv>

<Gn> → <Det> <Sa> <Nom> <Sa> <Relative>
<Relative> → qui <Gv> | ε
<Gv> → <Verbe> | <Verbe> <Gn>
<Sa> → <Adj> | ε

<Det> → un | le
<Nom> → chat | rat
<Verbe> → aime | mange
<Adj> → beau | féroce
```

La phrase « le chat féroce aime un beau rat qui mange » est engendrée, à partir de l'axiome <Phrase>, par cette grammaire. On remarque qu'il en est de même pour la phrase « le chat qui aime le chat qui aime le chat qui aime le chat qui aime aime ».

Cette remarque illustre bien une des difficultés de la modélisation d'une langue naturelle. C'est l'usage de la récursivité qui permet de donner une description finie d'un ensemble infini de mots. Il se trouve que, pour une langue naturelle, on a envie d'ajouter des critères esthétiques pour restreindre l'usage de la récursivité.

3.2 Classes de langages

Dans ce paragraphe nous convenons que a et b désignent des terminaux, A et B des non terminaux , α et β des mots de $(N \cup V)^*$.

1. On remarquera que ce vocabulaire est conçu en fonction d'un seul genre et d'un seul nombre. Cette propriété explique le fait que la grammaire ne comporte pas de règle d'accord.

3.2.1 Classification des grammaires

Les grammaires peuvent être plus ou moins compliquées. Une hiérarchie a été proposée par CHOMSKY. Cette classification dépend de la forme des règles. Pour des raisons difficiles à expliquer à ce niveau de l'exposé, les dérivation vides $A \rightarrow \varepsilon$ sont exclues des types 1 à 3, avec une seule exception, l'éventuelle règle $S \rightarrow \varepsilon$, nécessaire quand le langage contient le mot vide; dans ce cas, on impose à S de ne jamais apparaître en partie droite de règle.

Définition 3.5 Hiérarchie de CHOMSKY

- Toutes les grammaires sont de type 0.

Si on admet la seule exception citée plus haut:

- Toutes les grammaires dont les règles $\alpha \rightarrow \beta$ vérifient $|\alpha| \leq |\beta|$ sont de type 1.
- Toutes les grammaires dont les règles sont de la forme $A \rightarrow \beta$ sont de type 2.
- Toutes les grammaires dont les règles sont de la forme $A \rightarrow a$ ou de la forme $A \rightarrow aB$ sont de type 3.

On constate une inclusion de ces diverses classes.

La restriction introduite sur les dérivations vides n'est pas, en pratique, strictement nécessaire pour les classes 2 et 3 car on peut toujours éliminer les dérivations vides pour trouver une grammaire équivalente du même type. La restriction vise principalement à simplifier la démonstration de l'inclusion des classes. Nous utilisons donc indifféremment les définitions:

Définition 3.6 Une grammaire de type 2 est une grammaire dont les règles sont de la forme $A \rightarrow \beta$, où β est quelconque.

Définition 3.7 Une grammaire de type 3 est une grammaire dont les règles sont de la forme $A \rightarrow \varepsilon$ ou $A \rightarrow a$ ou encore $A \rightarrow aB$.

3.2.2 Classes de langages

Définition 3.8 Un langage est dit de type i s'il peut être décrit par une grammaire de type i.

On constate également une inclusion de ces diverses classes de langage. Un langage de type i « strict » est un langage qui peut être décrit par une grammaire de type i et qui ne peut pas être décrit par une grammaire de type i + 1. Nous serons parfois amenés à prendre des libertés avec cette définition en omettant, si le contexte s'y prête, le qualificatif strict.

3.2.3 Quelques paradigmes

Langages de type 3

- identificateurs Pascal
- constantes entières

- $\{a^n b^n, n \leq 6\}$

Langages de type 2

- $\{a^n b^p, n > p\}$
- $\{a^n b^n, n \geq 0\}$
- L'ensemble des programmes Pascal tels que le compilateur ne signale pas d'erreurs dites « de syntaxe ».

Langages de type 1

- $\{a^n b^n c^n, n \geq 0\}$

Il est plus difficile de donner des exemples de langages de type 0 et de langages qui ne sont même pas de type 0 (on ne peut pas les décrire par une grammaire).

Considérons l'ensemble des textes de programme Pascal corrects. On peut numéroter ces programmes p_0, p_1… (par exemple en rangeant les textes par tailles croissantes, l'ordre lexicographique étant utilisé pour classer deux programmes dont les textes ont la même taille). A l'aide de ces programmes on peut définir deux langages :

- $L = \{a^n \mid$ le programme p_n s'arrête lorsqu'on l'exécute en lui fournissant la donnée $n\}$
- $L' = \{a^n \mid$ le programme p_n ne s'arrête pas lorsqu'on l'exécute en lui fournissant la donnée $n\}$

On admettra que L est un langage de type 0 alors que L' n'est même pas de type 0.

3.3 Appartenance au langage engendré par une grammaire

Un automate est un algorithme (une machine) prenant en entrée un mot et donnant une réponse oui (l'automate accepte le mot) ou non (l'automate refuse le mot). Un automate est donc un moyen de caractériser un langage (les mots acceptés). On dit qu'une grammaire est un procédé *génératif* alors qu'un automate fonctionne en *reconnaissance*.

Un des problèmes liés à la notion d'automate, c'est celui qui consiste à savoir si l'automate donne toujours une réponse oui/non en un temps fini.

3.3.1 Grammaires de type 1

Propriété 3.1 Étant donné une grammaire G de type 1 et un mot m on peut décider en un temps fini si $m \in L(G)$ ou si $m \notin L(G)$.

Une grammaire de type 1 n'ayant pas de règle capable « raccourcir » un mot lors d'une réécriture il est possible de calculer l'ensemble T contenant tous les mots de $(N \cup V)^*$ qui dérivent de l'axiome et qui ont une longueur inférieure ou égale à |m|. Il suffit alors de tester l'appartenance ou non de m à T.

Algorithme

1. Poser $T_1 = \{S\}$
2. Itérer le calcul de T_i jusqu'à ce que $T_i = T_{i-1}$
   $T_i = \{\alpha \in (N \cup V)^* \mid \exists \beta \in T_{i-1}, \beta \Rightarrow_G \alpha \wedge |\alpha| \leq |m|\}$
4. Soit T le résultat final, on a: $m \in L(G) \Leftrightarrow m \in T$

L'arrêt de l'algorithme est assuré par le fait que lorsque $\beta \Rightarrow_G \alpha$ on a, puisque G est de type 1, $|B| \leq |\alpha|$.

Exemple

On considère le mot m = abca et la grammaire G dont les règles sont:

S → a S B C	C B → B C	a B → a b
S → a B C		b B → b b
		b C → b c
		c C → c c

On obtient:

$T_1 = \{S\}$
$T_2 = \{S, aSBC, aBC\}$
$T_3 = \{S, aSBC, aBC, abC\}$
$T_4 = \{S, aSBC, aBC, abC, abc\}$
$T_5 = T_4$

D'où abca ∉ L(G).

3.3.2 Grammaires de type 0

Propriété 3.2 Étant donné une grammaire G de type 0 et un mot m on peut décider en un temps fini si $m \in L(G)$.

Il est en effet possible de calculer toutes les dérivations de longueur 1, puis toutes les dérivations de longueur 2, puis…

On admettra, en revanche, qu'il n'est pas toujours possible, dans le cas général, de déterminer en un temps fini si $m \notin L(G)$.

3.3.3 Grammaires de type 3

Notons Type3(V) la classe des langages descriptibles par une grammaire de type 3.

La classe des langages descriptibles par une grammaire de type 3 (que nous notons Type3(V)) et la classe des langages reconnaissables sont identiques.

3.3.3.1 Inclusion de la classe des langages descriptibles par une grammaire de type 3

dans la classe des reconnaissables

À toute grammaire G = (N, V, S, P) de type3 on peut associer un automate à nombre fini d'état M = (Q, V, δ, q_0, F) en posant:

- Q = N ∪ {f} avec f ∉ N
- B ∈ δ(A, a) si A → aB ∈ P
- f ∈ δ(A, a) si A → a ∈ P
- q_0=S
- F= {f} si S → ε ∈ P, {S, f} sinon

À toute dérivation S $\Rightarrow^*_G$ m on peut associer un chemin de trace m dans l'automate reliant q_0 à un état final et réciproquement, ce qui assure que L(G) = L(M).

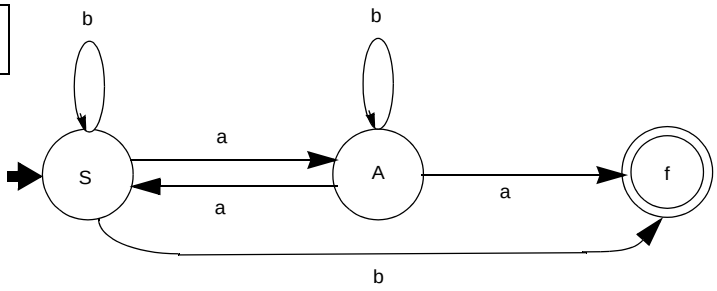
Donc Type3(V) ⊆ Rec(V).

Exemple

Règles de G

S → aA | bS | b

Automate associé



3.3.3.2 Inclusion de la classe des langages reconnaissables dans la classe des langages descriptibles par une grammaire de type 3

À tout automate à nombre fini d'états M = (Q, V, δ, q_0, F) dont l'état initial est final on peut associer un automate M' = (Q ∪ {q'_0, f'} , V, δ', q'_0, F - {q_0} ∪ {q'_0, f'}) équivalent tel qu'aucune transition n'aboutisse à l'état initial. Il suffit:

- d'ajouter une transition (q'_0, a, q) pour chaque transition (q_0, a, q): on simule les ancien « départs »;
- d'ajouter une transition (q, a, f') pour chaque transition (q, a, q_0): on simule les anciennes « arrivées gagnantes » en q_0.

À tout automate déterministe à nombre fini d'état M = (Q, V, δ, q_0, F), tel que si q_0 est final alors aucune transition n'aboutit en q_0, on peut associer une grammaire G = (N, V, S, P) de type 3 en posant:

- N = Q
- $q_i \rightarrow aq_j \in P$ si $\delta(q_j, a) = q_i$
- $q_i \rightarrow a \in P$ si $\delta(q_j, a) \in F$
- $q_0 \rightarrow \varepsilon \in P$ si $q_0 \in F$

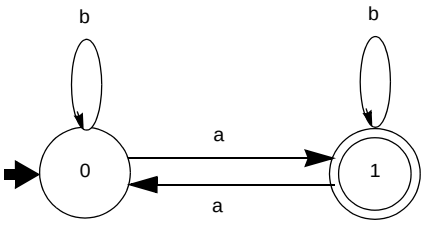
- $S = q_0$

À tout chemin de trace m dans l'automate reliant q_0 à un état final on peut associer une dérivation $S \Rightarrow^*_G m$ et réciproquement, ce qui assure que L(G) = L(M).

Donc Rec(V) ⊆ Type3(V).

Exemple

Automate M



Règles de la grammaire associée

$<q_0> \rightarrow b<q_0> \mid a<q_1>$
$<q_1> \rightarrow a<q_0> \mid b<q_1>$
$<q_0> \rightarrow a$
$<q_1> \rightarrow b$

3.3.3.3 Grammaires linéaires droites

La définition des grammaires de type 3 est très contraignante: elle permet d'affirmer qu'une grammaire de type 3 est une grammaire de type 1 et elle facilite certaines démonstrations en limitant le nombre de cas à envisager.

D'un point de vue pratique, il est comode d'utiliser des grammaires dites *linéaires droites.*

Définition 3.9 Une grammaire est dite linéaire droite si toutes ses règles sont de la forme $A \rightarrow mB$ ou $A \rightarrow m$, avec $m \in V^*$.

Le lecteur pourra montrer que, étant donné une grammaire linéaire droite, on peut toujours trouver une grammaire de type 3 équivalente.

3.4 Grammaire vs automates

Un automate est un algorithme (une machine) prenant en entrée un mot et donnant une réponse oui (l'automate accepte le mot) ou non (l'automate refuse le mot). Un automate est donc un moyen de caractériser un langage (les mots acceptés). On dit qu'une grammaire est un procédé *génératif* alors qu'un automate fonctionne en *reconnaissance.*

Les automates classiques pour les langages de type 3 (resp. 2) sont les automates à nombre fini d'états (resp. automates à pile). La théorie des langages permet d'obtenir certains résultats:

- équivalence grammaire/automate et algorithmes de traduction;
- déterminisation (dans certain cas), équivalence (toujours dans certains cas) ce qui permet de produire des analyseurs performants.

L'idée de base, du point de vue du génie logiciel, consiste donc à donner une grammaire respectant des contraintes plus ou moins fortes et à produire automatiquement un programme d'analyse (génération automatique d'analyseurs). Le procédé de traduction étant prouvé, les programmes ainsi produits sont corrects par construction.

CHAPITRE 4

Grammaires algébriques

4.1 Grammaires algébriques

4.1.1 Définition

Une *grammaire algébrique* est une grammaire dont les règles sont toutes de la forme $A \to \alpha$, $A \in V$, $\alpha \in (V \cup N)^*$.

Par rapport à une grammmaire de type 2 dans une grammaire algébrique on peut donc:

- utiliser sans restriction des règles de la forme $A \to \varepsilon$
- faire apparaître l'axiome en partie droite de règle, même si celui-ci se dérive en vide.

Une grammaire de type 2 est une grammaire algébrique et nous montrons plus loin qu'à toute grammaire algébrique il est possible d'associer une grammaire de type 2 équivalente (ie. engendrant le même langage).

4.1.2 Notations

Nous nous intéressons dans ce chapitre aux grammaires algébriques $G = (N, V, P, S)$ et nous utilisons les notations suivantes:

- G_A pour désigner la grammaire (N, V, P, A);
- $\alpha \Rightarrow_G^+ \beta$ pour $\alpha \Rightarrow_G^* \beta$ et $\alpha \neq \beta$ (réécriture en au moins un pas);
- $\alpha \Rightarrow_G^k \beta$ pour une réécriture en exactement k pas.

4.1.3 Une grammaire algébrique est une grammaire non contextuelle

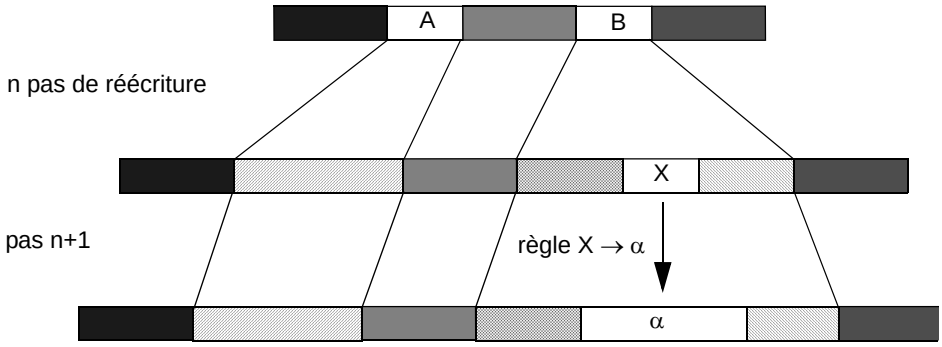
De par la définition, les non-terminaux d'une grammaire algébrique se réécrivent d'une manière indépendante du contexte.

Cette propriété est parfois appelée *lemme fondamental*.

Propriété 4.1 **Lemme fondamental**. Étant donné une grammaire algébrique G, si $m_1A_1...m_nA_nm_{n+1} \Rightarrow_G^* \alpha$ alors α s'écrit $m_1\alpha_1...m_n\alpha_nm_{n+1}$ et, pour chaque i, $A_i \Rightarrow_G^* \alpha_i$.

On démontre cette propriété par réccurence sur la longueur des dérivations.

- Elle est trivialement vraie pour les dérivations de longueur nulle.
- Si elle est vraie pour les dérivations de longueur n, elle est conservée lorsqu'on ajoute un pas de dérivation supplémentaire.



4.2 Arbres syntaxiques

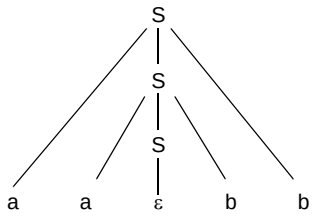
4.2.1 Définition et exemples

Définition 4.1 Un arbre $\mathcal{A}$ est un *arbre syntaxique* pour une grammaire algébrique G = (N, V, P, S) si et seulement si:

- Chaque nœud de $\mathcal{A}$ possède une étiquette qui est un symbole de $V \cup N$ ou ε.
- L'étiquette de la racine est S.
- Si un nœud a des descendants alors son étiquette est dans N.
- Si les nœuds $\mathcal{A}_1$,..., $\mathcal{A}_n$ sont les descendants directs (on les ordonne à partir de la gauche) d'un nœud étiqueté X et si la règle $A \rightarrow \Omega_1, ..., \Omega_n \in P$ alors chaque nœud $\mathcal{A}_i$ a pour étiquette Ω_i.

Exemple 1

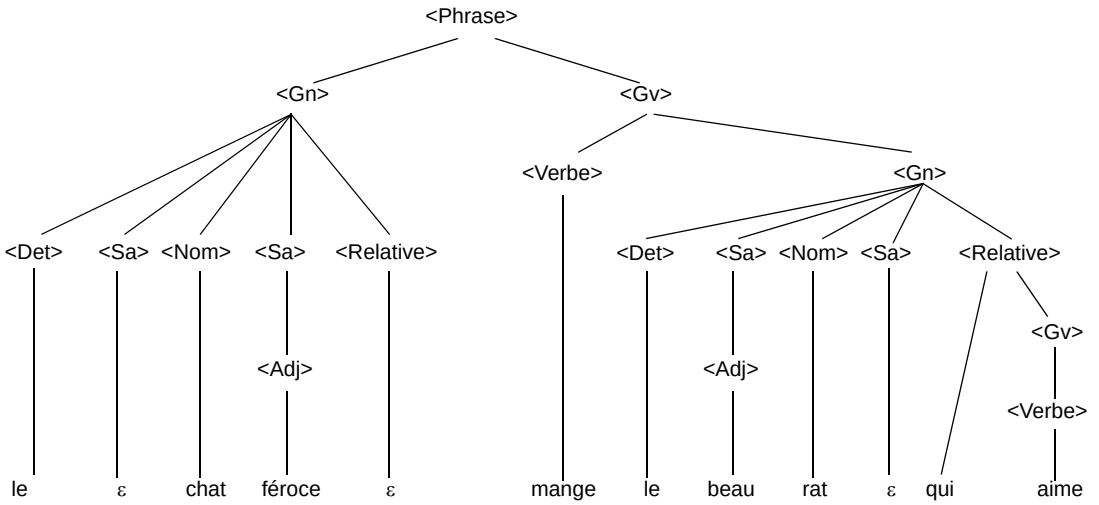
$S \rightarrow aSb \mid \varepsilon$



Exemple 2

<Phrase> → <Gn> <Gv>

<Gn> → <Det> <Sa> <Nom> <Sa> <Relative>
<Relative> → qui <Gv> | ε
<Gv> → <Verbe> | <Verbe> <Gn>
<Sa> → <Adj> | ε



Définition 4.2 Le *mot des feuilles* d'un arbre syntaxique est obtenu en concaténant ses feuilles (dans l'ordre de leur découverte par un parcours descendant de gauche à droite).

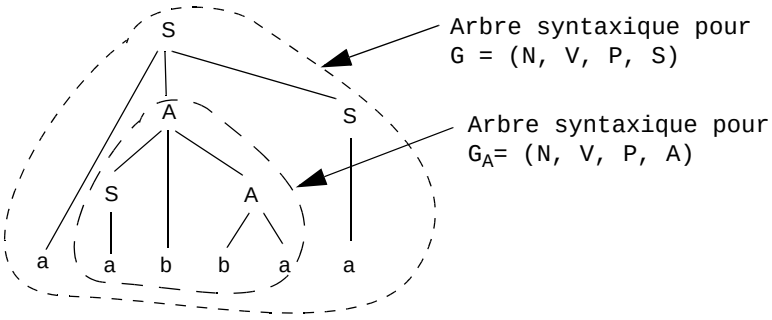
4.2.2 Propriétés

Propriété 4.2 Étant donné une grammaire algébrique G, un mot m appartient à L(G) si et seulement si m est le mot des feuilles d'un arbre syntaxique pour G.

On remarque qu'un sous arbre étiqueté par A d'un arbre syntaxique pour une grammaire G est un arbre syntaxique pour la grammaire G_A.

Exemple

$S \rightarrow a\,A\,S \mid a \qquad A \rightarrow S\,b\,A \mid b\,a \mid SS$

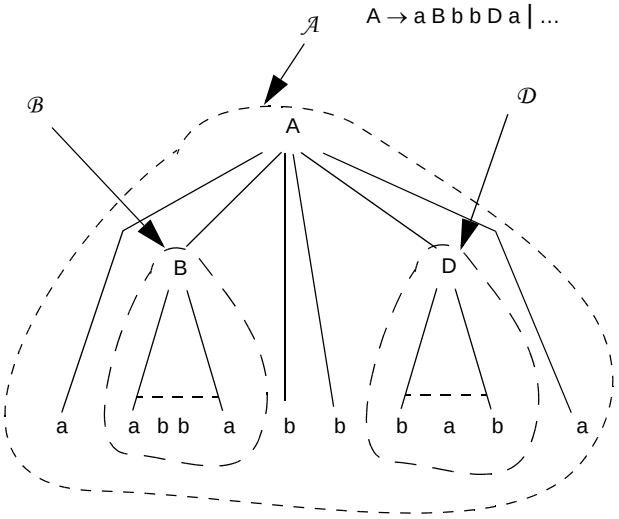


La propriété 4.2 est une conséquence d'une propriété plus générale.

Propriété 4.3 Étant donné une grammaire algébrique G, m est mot des feuilles d'un arbre syntaxique pour G_A si et seulement si $A \Rightarrow^*_{G_A} m$.

Pour démontrer que si m est mot des feuilles d'un arbre syntaxique $\mathcal{A}$ de racine A alors $A \Rightarrow^*_{G_A} m$ on procède, comme suggéré dans la figure suivante, par récurrence sur le nombre k de nœuds de $\mathcal{A}$ qui ne sont pas des feuilles.

Cas général : k > 1



$\mathcal{B}$ et $\mathcal{D}$ sont des arbres syntaxiques qui ont strictement moins de k nœuds qui ne sont pas des feuilles. Par récurrence, on peut donc dire que :

$B \Rightarrow^*_{G_B} abba$ et

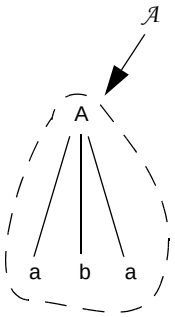
$D \Rightarrow^*_{G_D} bab$

Comme la règle $A \rightarrow a\,B\,b\,b\,D$ est dans P, on a :

$A \Rightarrow^*_{G_A} aBbbDa$

D'où

$A \Rightarrow^*_{G_A} aabbabbbaba$



Cas d'arrêt : k = 1

En dehors de la racine, $\mathcal{A}$ ne comporte que des terminaux.
Comme la règle A → a b a est dans P, on a: $A \Rightarrow^*_{G_A} aba$

Pour démontrer la réciproque, on procède par récurrence sur la longueur des dérivations.

- Lorsque la dérivation $A \Rightarrow^*_{G_A} m$ a pour longueur 1, la règle A → m est dans P. Comme m ne contient que des terminaux, il est mot des feuilles de l'arbre syntaxique dont la racine est étiquettée A et dont les descendants directs sont les symboles de m.
- Lorsque la dérivation $A \Rightarrow^*_{G_A} m$ a pour longueur n+1, on peut distinguer le premier pas de dérivation $A \Rightarrow^*_{G_A} \alpha$. Les non-terminaux X qui apparaissent alors dans α se récrivent en des sous-mots de m (cf. lemme fondamental) en au plus n pas de réécriture: par récurrence on peut leur associer des arbres syntaxiques et les utiliser pour construire, à partir de la règle A → α, un arbre syntaxique dont m est le mot des feuilles.

4.2.3 Propriétés

On appelle *dérivation gauche* de a en b, une séquence de dérivations $\alpha \Rightarrow_G^* \beta$ dans laquelle, à chaque pas, c'est le non-terminal le plus à gauche qui est réécrit.

Exemple

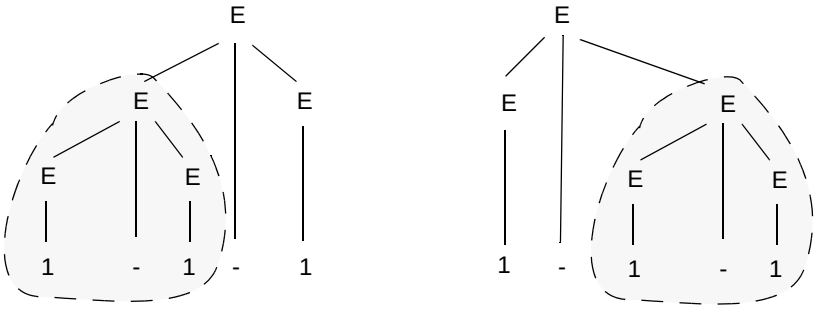
Considérons la grammaire G_1 = ({E}, {1, -, (,)}, {E → E - E | (E) | 1}, E) et les quatre dérivations ci-après.

D1	D2	D3	D4
E	E	E	E
E - E	E - **E**[a]	**E** - E	E - E
1 - E	E - **E - E**	**E - E** - E	E - 1
1 - E - E	E - 1 - E	1 - E - E	E - E - 1
1 - 1 - E	E - 1 - 1	1 - 1 - E	E - 1 - 1
1 - 1 - 1	1 - 1 - 1	1 - 1 - 1	1 - 1 - 1

a. Pour D2 et D3, nous utilisons la graisse pour préciser, lors du deuxième pas de dérivation, le non-terminal objet de la réécriture.

D1 et D3 sont des dérivations gauche, D2 et D4 ne le sont pas.

Le mot 1 - 1 - 1 appartient donc à $L(G_1)$. On peut lui associer deux arbres syntaxiques différents.



Définition 4.3 Une grammaire est dite ambiguë si on peut trouver, pour un même mot, deux arbres syntaxiques différents.

Une grammaire algébrique sert à décrire un langage et cette description permet en outre de munir les mots d'une structure. Lorsque la grammaire est ambiguë, certains mots se voient attribuer plusieurs structures possibles. La grammaire G_1 ci-dessus est donc ambiguë: le mot 1 - 1 - 1 possède deux arbres syntaxiques qui correspondent aux structures [1 - 1] - 1 et 1 - [1 - 1].

Dans ce cas particulier il est facile de trouver deux grammaires G_2 et G_3, équivalentes à G_1, qui ne sont pas ambiguës.

- G_2 = ({E, T}, {1, -, (,)}, {E → E - T | T, T → (E) | 1}, E)
- G_3 = ({E, T}, {1, -, (,)}, {E → T - E | T, T → (E) | 1}, E)

D4	D5	D6	D7
E	E	E	E
E - T	E - T	T - E	T - E
E - T - T	E - 1	1 - E	T - T - E
T - T - T	E - T - 1	1 - T - E	T - T - T
1 - T - T	E - 1 - 1	1 - 1 - E	T - T - 1
1 - 1 - T	T - 1 - 1	1 - 1 - T	T - 1 - 1
1 - 1 - 1	1 - 1 - 1	1 - 1 - 1	1 - 1 - 1

Sur cet exemple on constate qu'une grammaire non-ambiguë peut être remarquablement porteuse d'information: les règles usuelles d'associativité du moins sont reflétées par les arbres syntaxiques de G2.

4.2.4 Arbres syntaxiques de type 2 et de type 3

On constate que:

- Les arbres syntaxiques d'une grammaire de type 2 stricte sont bien des arbres, au sens plein du terme.
- Les arbres syntaxiques d'une grammaire de type3 stricte sont des cas particuliers d'arbres: des listes.

Cette distinction, profonde, fait que l'analyse des langages de type 3 (automates à nombre fini d'états) est relativement facile.

4.3 Grammaires algébriques et programmes

4.3.1 Tirer des dérivations au hasard

Étant donné un langage décrit par une grammaire algébrique, nous souhaitons écrire un programme qui, lorsqu'il est exécuté, produit en résultat un mot du langage « tiré au hasard ».

Le sens de l'expression « tiré au hasard » reste à préciser. Remarquons en premier lieu que si le langage est infini il est impossible d'avoir un tirage équiprobable. Ce que nous recherchons pour notre programme, c'est un comportement externe humainement acceptable: on ne doit pas pouvoir prévoir (du moins avec un raisonnement simple) le résultat de la prochaine exécution.

Pour simuler un tel comportement, nous choisissons, pour chaque exécution, de tirer au hasard une dérivation gauche à partir de l'axiome. Lorsque plusieurs réécriture sont possibles, on aura, au préalable, fixé leurs probabilités respectives d'être sélectionnées.

À titre d'exemple considérons la grammaire G dont les règles sont:

- S → a A {1} | b B {1} | ε {2}
- A → b S {2} | a A A {1}
- B → a S {2} | b B B {1}

Les nombres entre accolades sont des annotations qui décorent la grammaire: on parle de *grammaire décorée*. Nous esquissons une explication de leur signification sur deux exemples:

- L'annotation {1} de la règle S → b B {1} signifie que, lorsqu'un S est réécrit, il est réécrit en b B une fois (1 est l'annotation de la partie droite b B) sur quatre (4 est la somme des annotations des parties droites possibles pour S).
- L'annotation {2} de la règle A → b S {2} signifie que, lorsqu'un A est réécrit, il est réécrit en b S deux fois (2 est l'annotation de la partie droite b S) sur trois (3 est la somme des annotations des parties droites possibles pour A).

À partir de la grammaire décorée, il est facile de produire automatiquement le programme qui suit.

```
public class MotsHasardeux {
    static int lancerDé (int n) {
         return (int) Math.round(Math.random()*(n - 1)) + 1 ;
    }
    public static String réécrireS () {
         // regles : S -> a A {1} | b B {1} | vide {2}
         switch (lancerDé(4)) {
         case 4 :
              return "a" + réécrireA () ;
         case 3 :
              return "b" + réécrireB () ;
         case 2 :
         case 1 :
              return "" ;
         default :
              return "" ;
         }
    }
    static String réécrireA () {
         // regles : A -> b S {2} | a A A {1}
         switch (lancerDé(3)) {
         case 3 :
         case 2 :
              return "b" + réécrireS () ;
         case 1 :
              return "a" + réécrireA () + réécrireA () ;
         default :
              return "" ;
         }
    }
    static String réécrireB () {
         // regles : B -> a S {2} | b B B {1}
         switch (lancerDé(3)) {
         case 3 :
         case 2 :
              return "a" + réécrireS () ;
         case 1 :
              return "b" + réécrireB () + réécrireB () ;
         default :
              return "" ;
         }
    }
}
```

4.3.2 Grammaires décorées et programmes

On constate que le programme ci-dessus étant une quasi paraphrase de la grammaire G décorée de ses annotations. Il est donc tout à fait possible de concevoir un traducteur transformant une grammaire décorée en un program-

me exécutable. Ce programme est capable de *générer* des mots appartenant à un langage donné.

Au chapitre suivant, nous nous posons la question de traduire une grammaire algébrique en un programme effectuant la *reconnaissance* de mots. Moyennant certaines restrictions sur la forme de la grammaire, de telles traduction sont possibles et les traducteurs utilisés s'appellent des *générateurs d'analyseurs syntaxiques.*

4.4 Quelques propriétés des grammaires algébriques

4.4.1 Élimination des dérivations vides

4.4.2 Suppression des non-terminaux superflus

Déterminer si L(G) est vide

Suppression des non-terminaux non productifs

Suppression des non-terminaux non accessibles

4.4.3 Élimination des règles de la forme $A \rightarrow B$

4.5 Formes normales

4.5.1 Forme normale de CHOMSKY

Définition 4.4 Une grammaire algébrique est en forme normale de Chomsky si toutes ses régles relèvent d'une des 2 formes suivantes:

- $A \rightarrow B\ C$
- $A \rightarrow a$

4.5.2 Forme normale de GREIBACH

Définition 4.5 Une grammaire algébrique est en forme normale de Greibach si toutes ses régles relèvent d'une des 2 formes suivantes:

- $A \rightarrow a\ B_1 \ldots B_n$
- $A \rightarrow a$

4.6 Lemme d'itération pour les langages algébriques

Propriété 4.4 **Théorème « uvwxy »**. Pour tout langage algébrique L, il existe 2 constantes p et q telles que, si $z \in L$ et si $|z| > p$ alors z peut s'écrire $z = u\,v\,w\,x\,y$, avec $|v\,w\,x| \leq q$ et $v\,x \neq \varepsilon$, de façon que, pour tout n, $u\,v^n\,w\,x^n\,y \in L$.

CHAPITRE 5

Analyse LL(1)

Nous nous intéressons dans ce chapitre aux grammires $G = (N, V, P, S)$ de type 2 et nous utilisons les notations suivantes:

- G_A pour désigner la grammaire (N, V, P, A);
- $\alpha \Rightarrow_G^+ \beta$ pour $\alpha \Rightarrow_G^* \beta$ et $\alpha \neq \beta$ (réécriture en au moins un pas);
- $\alpha \Rightarrow_G^k \beta$ pour une réécriture en exactement k pas.

5.1 Présentation informelle

5.1.1 Analyse LL(1)

Étant donné une grammaire algébrique G, réaliser l'analyse syntaxique d'un mot $m = a_1 \ldots a_n$ c'est:

- construire l'arbre syntaxique de m si m est dans $L(G)$;
- produire des diagnostics d'erreur adéquats dans le cas contraire.

Pour construire l'arbre syntaxique de m on peut chercher à construire une dérivation gauche $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_r = m$ en partant de l'axiome. Si cette construction est possible, la situation au pas i de la construction est la suivante:

- $\alpha_i = a_1 \ldots a_j A b$
- $m = a_1 \ldots a_j \ldots a_{j+k} \ldots a_n$
- les A-règles sont $A \rightarrow \beta_1 \mid \ldots \mid \beta_s$

Si la connaissance des k symboles $a_{j+1} \ldots a_{j+k}$ suffit toujours pour choisir parmi les règles $A \rightarrow \beta_i$ celle qui doit être utilisée au prochain pas de dérivation on dit que la grammaire G est LL(k)[1]. Il s'agit d'une analyse:

- descendante (on construit la dérivation à partir de l'axiome)
- de gauche à droite (on construit une dérivation gauche)
- déterministe sur prélecture de k symboles (la connaissance des k symboles $a_{j+1}...a_{j+k}$ suffit toujours pour déterminer la dérivation suivante).

Nous nous intéressons, dans ce chapitre, aux grammaires LL(1).

À titre d'exemple, considérons la grammaire G dont les règles sont {S → a S b | ε} qui est est LL(1) :

- si le symbole de prélecture est un a on utilise la première règle;
- si le symbole de prélecture est un b, on utilise la seconde règle.

Exemple

Construction de la dérivation	Progression de la tête de lecture	Règle sélectionnée
S	**a**aabbb	1
aSb	a**a**abbb	1
aaSbb	aa**a**bbb	1
aaaSbbb	aaa**b**bb	2
aaabbb	aaabbb	le mot est reconnu

5.1.2 Automate procédural

À toute grammaire LL(1) on peut associer un automate que nous qualifions d'*automate procédural*. Pour un automate à nombre fini d'états on ne peut mémoriser plus informations sur l'état d'analyse qu'il n'y a d'états. Pour un automate procédural, la mémorisation s'effectue dans la pile des appels de procédure: la capacité de mémorisation n'est donc pas, au plan théorique, bornée. Au plan pratique il est évident que la pile des appels a une taille maximum.

Un automate procédural comporte un dispositif de lecture du mot à analyser et un ensemble de procédures.

- Le dispositif de lecture de la bande d'entrée, comporte une tête de lecture LexLu qui désigne la première unité lexicale (celles qui précédent sont perdues). Lexlu contient donc le symbole qui permet de choisir entre les diver-

1. L'abbréviation LL provient de l'anglais *from* L*eft to right using* L*eftmost derivation.*

ses possibilités de dérivation. La procédure AvTête permet de faire avancer la tête de lecture sur l'unité lexicale suivante.

- À chaque non terminal de la grammaire on associe une procédure du même nom. On dispose d'un procédé de construction automatique pour construire les procédures à partir de la grammaire.

Pour le cas de la grammaire G évoquée au 5.1.1, comme il n'y a qu'un seul non-terminal il n'y a qu'une seule procédure. Nous donnons ci-après son code, de manière à disposer d'un exemple pour illustrer le fonctionnement d'un automate procédural. Le procédé de construction est évoqué au paragraphe suivant.

```
proc S
    cas Lexlu dans
        qd Lexlu = 'a' →
            AvTete ;
            S ;
            si Lexlu <> 'b' alors erreur[1] fsi ;
            AvTete ;
        qd Lexlu = 'b' →
        aut
            erreur
    fcas
```

Nous donnons une trace d'exécution de cet automate pour analyser le mot aaabbb.

Appel initial de S	Appel 2	Appel 3	Appel 4	Bande d'entrée
LexLu vaut 'a' Avtete *Appel de S*				**a**aabbb
	LexLu vaut 'a' Avtete *Appel de S*			a**a**abbb
		LexLu vaut 'a' Avtete *Appel de S*		aa**a**bbb
			LexLu vaut 'b' *Fin de S*	aaa**b**bb

1. Le traitement des erreurs peut être plus sophistiqué: dans ce cas on sait qu'un 'b' aurait dû être présent sur la bande d'entrée alors que, pour l'erreur suivante, on constate la présence d'un caractère inattendu sur la bande d'entrée. La procédure erreur peut être munie de paramètres adéquats.

Appel initial de S	Appel 2	Appel 3	Appel 4	Bande d'entrée
		LexLu vaut 'b' Avtete *Fin de S*		aaa**b**bb
	LexLu vaut 'b' Avtete *Fin de S*			aaab**b**b
LexLu vaut 'b' Avtete *Fin de S*				aaabb**b**

Lors de l'exécution de l'appel 4, la pile contient 3 appels de S en attente de retour correspondant à la détection d'un 'a' sur la bande d'entrée. Lors des retours on réalise un appariement des 'a' rencontrés dans la première partie de l'analyse avec les 'b' de la seconde partie du mot.

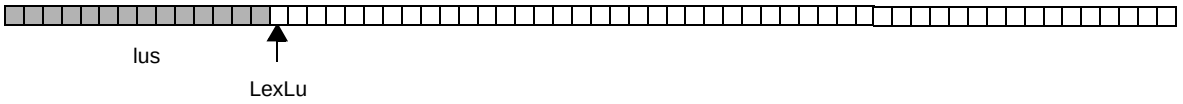
5.2 Automate procédural

5.2.1 Vers une traduction systématique de la grammaire

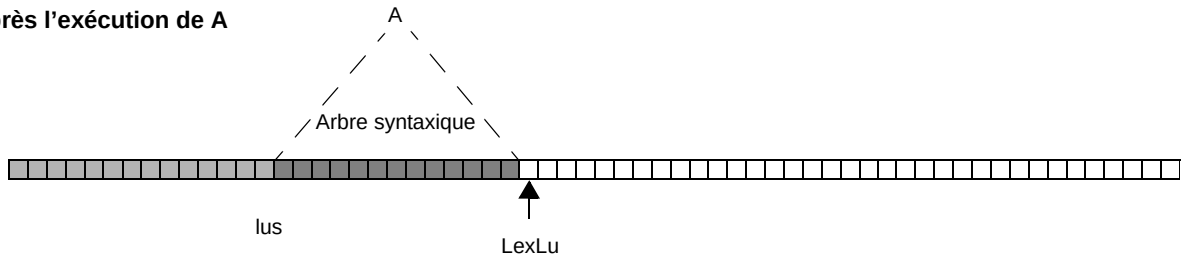
5.2.1.1 Structure générale

À chaque non-terminal A on associe une procédure A sans paramètre, capable de reconnaître un mot de $L(G_A)$ en préfixe de la bande d'entrée.

Avant l'exécution de A



Après l'exécution de A



La structure générale de la procédure A est la suivante (les A-règles sont $A \rightarrow \beta_1 \mid \ldots \mid \beta_s$).

proc A

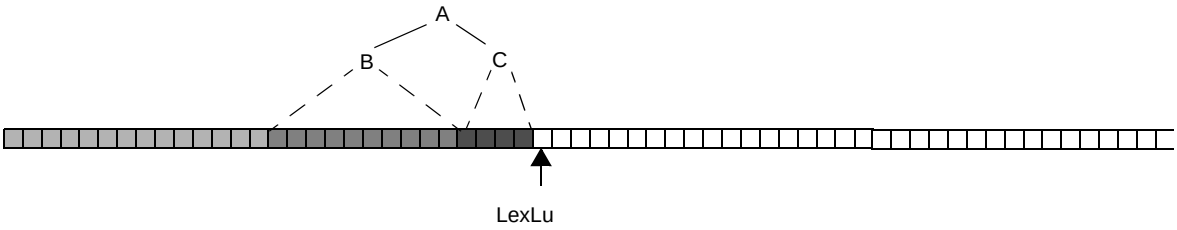
```
cas Lexlu dans
      qd <sélection 1> → <traitement β1> ;
      …
      qd <sélection n> → <traitement βn> ;
      aut
            erreur
fcas
```

5.2.1.2 Traitement des parties droites

Nous illustrons le traitement des parties droites à l'aide de trois exemples que le lecteur pourra sans peine généraliser.

A → B C

L'arbre synatxique construit pour A doit comporte deux sous-arbres de racines respectives B et C.

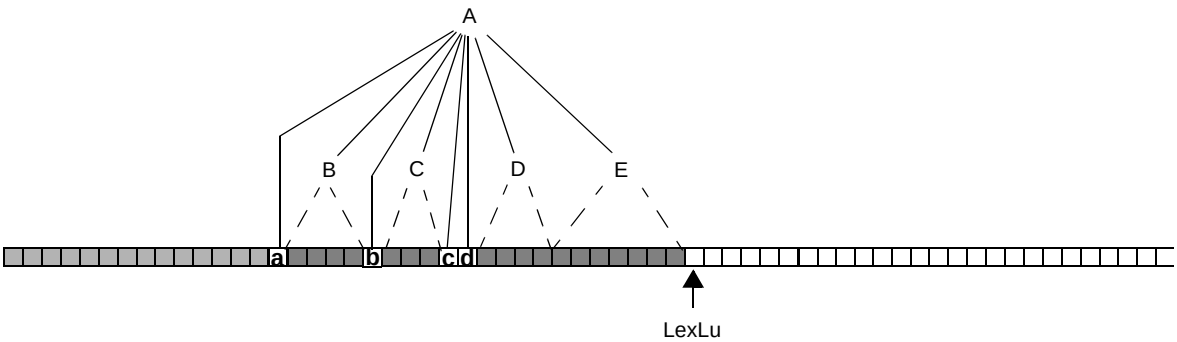


D'où:

```
-- traitement de la règle A -> B C
    B ;
    C ;
```

A → a B b C c d D E

Le traitement d'une telle règle prend en compte les arbres syntaxiques des non-terminaux de la partie gauche et vérifie que, dans le mot des feuilles, les terminaux prennent leur place.



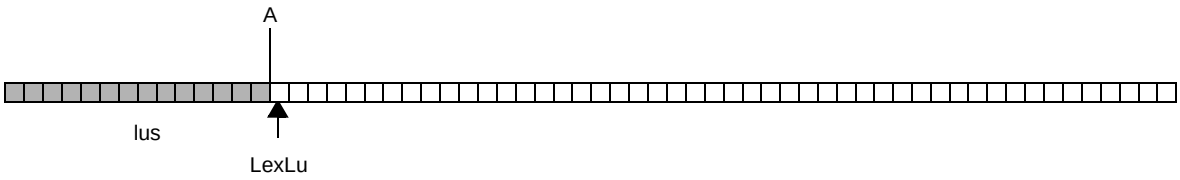
D'où:

```
-- traitement de la règle A -> a B b C c D E
    AvTete ;  -- si on a sélectionné cette règle il est obli-
gatoire qu'un a soit
```

```
                -- sous la tête de lecture
B ;
si LexLu ≠ b alors erreur fsi ; AvTete ;
C ;
si LexLu ≠ c alors erreur fsi ; AvTete ;
D
E ;
```

A $\rightarrow \varepsilon$

Le traitement d'une telle règle consiste à ne rien faire !



D'où:

```
-- traitement de la règle A -> vide
    -- on ne fait rien
```

5.2.1.3 Traitement des sélections

Appelons Premiers(β) l'ensemble des terminaux qui peuvent apparaître à gauche d'une quelconque séquence de dérivations à partir de β.

$$\text{Premiers}(\beta) = \{a | \beta \underset{G}{\overset{*}{\Rightarrow}} a\omega\}$$

À l'exception de la règle $A \rightarrow \varepsilon$, pour sélectionner une règle $A \rightarrow \beta$ parmi les A-règles, il est nécessaire que LexLu appartienne à Premiers(β).

Appelons Suivants(A) l'ensemble des terminaux qui peuvent apparaître juste à droite de A dans une quelconque dérivation à partir de l'axiome.

$$\text{Suivants}(A) = \{a | S \underset{G}{\overset{*}{\Rightarrow}} \omega_1 A a \omega_2\}$$

Pour sélectionner parmi les A-règles une règle $A \rightarrow \beta$ telle que β se dérive en vide, il est nécessaire que LexLu appartienne à Suivant(A).

Losqu'on impose certaines conditions sur la grammaire (cf. paragraphe suivant), ces conditions permettent de sélectionner sans ambiguïté une seule règle, d'où:

```
-- sélection de la règle A -> β, cas où β ne se dérive jamais
en vide
    LexLu apt Premiers(β)

-- sélection de la règle A -> β, cas où β peut se dériver en
vide
```

`LexLu` **apt** `Premiers(β)` **ou** `LexLu` **apt** `Suivant(A)`

5.2.2 Conditions pour qu'une grammaire soit LL(1)

Définition 5.1 Une grammaire est LL(1) si, pour chaque non-terninal A dont les A-règles sont $A \rightarrow \beta_1 \mid ... \mid \beta_s$, les 3 conditions suivantes sont vérifiées:

- Pour $i \neq j$, $Premier(\beta_i) \cap Premiers(\beta_j) = \varnothing$
- Il y a au plus une partie droite capable de se dériver en vide
- Si β_n peut se dériver en vide, alors pour $i \neq n$, $Premier(\beta_i) \cap Suivants(A) = \varnothing$

On note Null(β) la fonction qui indique si un mot β est capable (ou non) de se dériver en vide.

Losqu'une grammaire est LL(1) les principes de sélection donnés au paragraphe précédent sont suffisants pour sélectionner sans ambiguïté la seule règle capable de prolonger la dérivation en cours.

Dans les trois exemples qui suivent, lorsqu'un a est sous la tête de lecture il est impossible de choisir entre les deux A-règles.

Condition 1	Condition 2	Condition 3
A → B C | D E D → a F | d B → a | b G	S → A B **A → F G |** ε B → a D F → c X | ε G → b Z | ε	S → A B **A → a C** | ε B → a D

On remarque qu'une grammaire comportant une règle récursive à gauche n'est pas LL(1). Ce cas de figure est assez fréquent pour proposer un procédé systématique d'élimination de la récursivité à gauche.

Règle récursive à gauche	À remplacer par
$A \rightarrow A\ \beta \mid \alpha$	$A \rightarrow \alpha\ R_A$ $R_A \rightarrow \beta\ R_A \mid \varepsilon$

5.3 Une étude de cas : les expressions arithmétiques

5.3.1 Grammaire LL(1) pour les expressions arithmétiques

Considérons une grammaire d'expressions infixées avec deux niveaux de priorité.

```
G1 :
    E → E Opadd T | T
    T → T Opmul F | F
    F → Idf | ( E )

    Opadd → + | -
    Opmul → * | :
    Idf → a | b
```

Cette grammaire n'est pas LL(1) : la condition 1 n'est pas respectée par les deux règles récursives à gauche. On peut éliminer ces récursivités.

```
G2 :

    E → T R_E
    R_E → Opadd T R_E | ε
    T → F R_T
    R_T → Opmul F R_T | ε
    F → Idf | ( E )

    Opadd → + | -
    Opmul → * | :
    Idf → a | b
```

Cette grammaire est LL(1), puisque:

```
Suivants(R_E) = {), fin}Premiers(Opadd T R_E) = {+, -}
Suivants(R_T) = {+, -, ), fin}Premiers(Opmul F R_T) = {*, :}
```

5.3.2 Automate procédural

On peut déduire de cette grammaire l'automate procédural suivant:

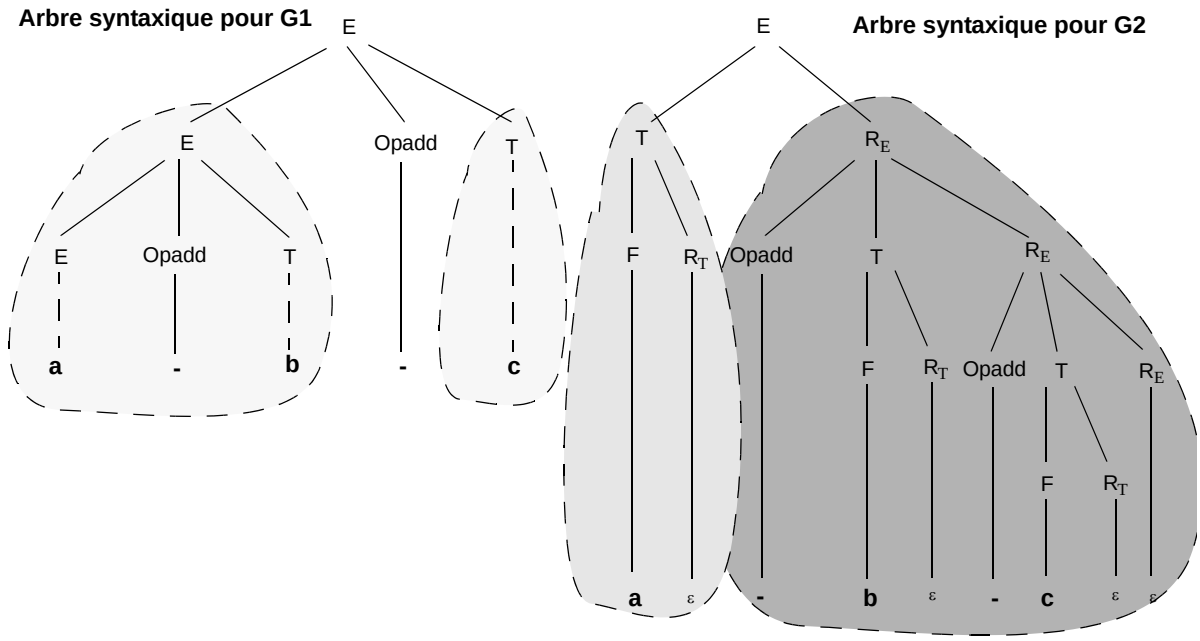
```
proc E
    cas Lexlu dans
          qd LexLu apt {a, b, (} → T ; R_E ;
          aut erreur
    fcas

proc R_E
    cas Lexlu dans
          qd LexLu apt {+, -} → Opadd ; T ; R_E ;
          qd LexLu apt {), fin} →
          aut erreur
    fcas

proc T
```

```
    cas Lexlu dans
        qd LexLu apt {a, b, (} → F ; R_T ;
        aut erreur
    fcas

proc R_T
    cas Lexlu dans
        qd LexLu apt {*, :} → Opmul ; F ; R_T ;
        qd LexLu apt {+, -, ), fin} →
        aut erreur
    fcas

proc F
    cas Lexlu dans
        qd LexLu apt {a, b} → Idf;
        qd LexLu apt {(} →
                    AvTete ;
                    E ;
                    si LexLu ≠ ) alors erreur fsi ;
                    AvTete ;
        aut erreur
    fcas

proc Opadd
    cas Lexlu dans
        qd LexLu apt {+, -} → AvTete
        aut erreur
    fcas

proc Opmul
    cas Lexlu dans
        qd LexLu apt {*, :} → AvTete
        aut erreur
    fcas

proc Idf
    cas Lexlu dans
        qd LexLu apt {a, b} → AvTete
        aut erreur
    fcas
```

5.3.3 Problèmes

Le remplacement d'une grammaire algébrique par une grammaire LL(1) équivalente (si c'est possible) peut avoir une conséquence dommageable: les arbres syntaxiques obtenus avec la nouvelle grammaire n'ont pas obligatoirement une

structure respectant les souhaits du concepteur de la grammaire de départ, comme en témoigne la figure ci-après.



Sur cet exemple, l'arbre syntaxique pour G2 est structurellement plus éloigné de l'arbre abstrait correspondant à l'expression analysée que celui de G1: l'analyse sémantique sera donc plus complexe.

L'absence de récursivité à gauche rend les grammaires LL(1) peu comodes pour analyser les langages d'expression: l'associativité à gauche des opérateurs est traditionnellement plus répandue que l'associativité à droite. C'est une des raisons qui expliquent le succès d'un autre mode d'analyse, l'analyse LALR.

5.4 Classe des langages LL(1)

5.4.1 Principaux résultats

La question se pose de savoir s'il est toujours possible de trouver une grammaire LL(1) équivalente à une grammaire algébrique donnée. Ce n'est pas toujours possible.

Propriété 5.1 Il existe des langages algébriques qui, pour aucun k, ne sont LL(k).

Pour illustrer cette propriété on peut considérer le langage des palindromes sur {a, b}, décrit par la grammaire G qui suit.

G : S → a S a | b S b | a | b | ε

Pour une valeur donnée de k, il est, dans le cas général, impossible de déterminer avec une prélecture de k symboles si on est, ou non, arrivé « au milieu » du mot.

Il reste donc à s'interroger sur la capacité de déterminer si un langage donné est descriptible par une grammaire LL(1). Nous admettrons la propriété qui suit:

Propriété 5.2 L'appartenance d'un langage à la classe des langages analysables par une grammaire LL(1) est indécidable.

5.4.2 Déterminer si une grammaire est LL(1)

Étant donné une grammaire algébrique G, nous cherchons à déterminer si elle est LL(1). Il s'agit donc de calculer la fonction Null et les ensembles Premiers et Suivants.

Calcul de la fonction Null

Nous avons vu au chapitre qui précède comment déterminer si un non-terminal A se dérive en vide. Pour qu'un mot α puisse se dérive en vide, il faut qu'il ne soit composé que de non-terminaux tous capables de se dériver en vide. Nous disposons donc d'un algorithme pour calculer la fonction Null.

Calcul des ensembles Premiers

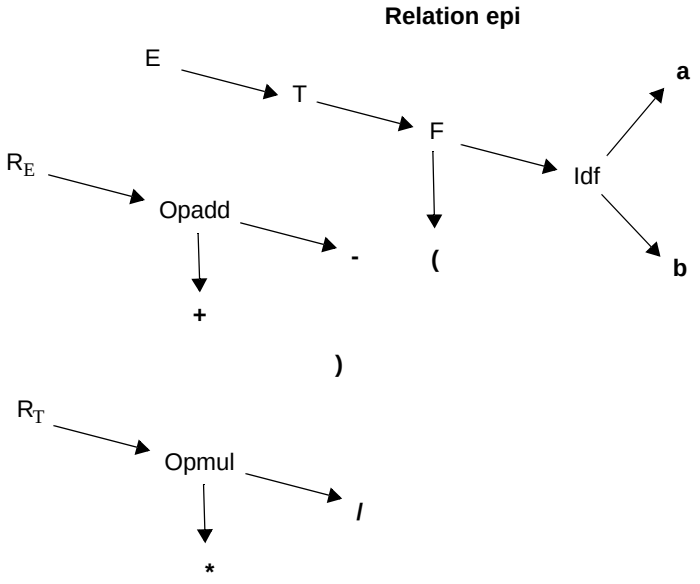
Posons $U = V \cup N$. On définit une relation binaire **epi** (est un premier immédiat) sur U

- $X \textbf{ epi } u = (X \to u\alpha \in P) \vee (X \to \beta u\alpha \in P \wedge Null(\beta))$

Les ensembles premiers se déduisent de la fermeture transitive de la relation **epi** et de la fonction Null.

$E \rightarrow T\ R_E$
$R_E \rightarrow Opadd\ T\ R_E \mid \varepsilon$
$T \rightarrow F\ R_T$
$R_T \rightarrow Opmul\ F\ R_T \mid \varepsilon$
$F \rightarrow Idf \mid (\ E\ )$

$Opadd \rightarrow + \mid -$
$Opmul \rightarrow * \mid :$
$Idf \rightarrow a \mid b$

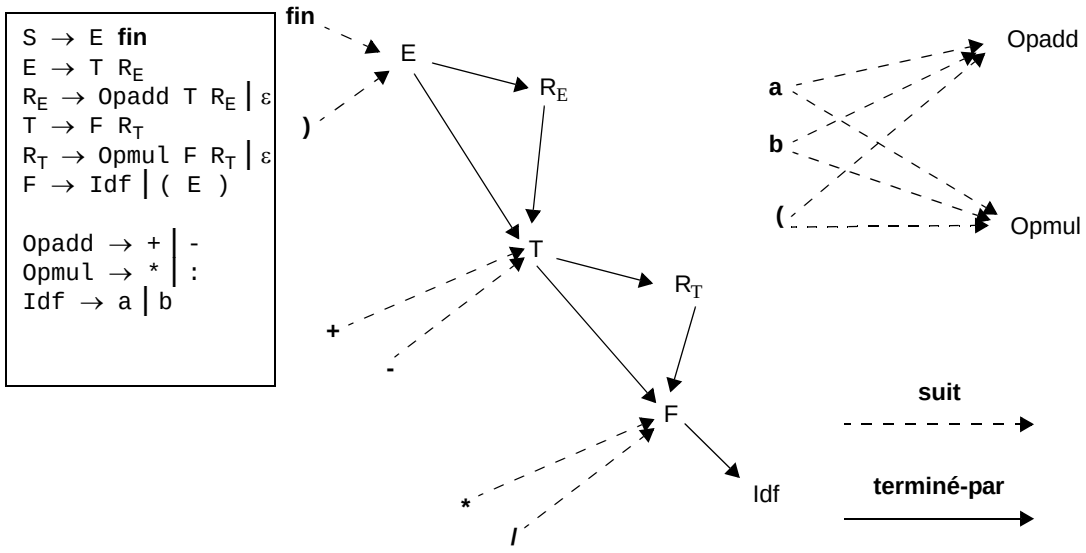


Calcul des ensembles Suivants

On définit trois relations:

- Y **a-droite** X lorsqu'il existe une règle $A \rightarrow \alpha XY\beta$
- a **suit** X, lorsqu'il existe une règle $A \rightarrow \alpha Xa\beta$ ou lorsque $a \in$ Premiers(Y) et que Y **a-droite** X
- X **terminé-par** Y lorsqu'il existe une règle $X \rightarrow \alpha Y$ ou lorsqu'il existe une règle $X \rightarrow \alpha Y\beta$, β pouvant se dériver en vide.

La fermeture transitive de la composition des relations **suit** et **terminé-par** permet de calculer les ensembles Suivants.



5.4.3 Heuristiques d'élimination des défauts

Nous donnons trois techniques de base utiles lorsqu'on cherche à modifier une grammaire pour trouver une grammaire LL(1) équivalente.

Il faut cependant être conscient qu'il n'existe pas d'algorithme général permettant la transformation.

Technique	Problème sur les A-règles	Remplacement
Élimination de la récursivité à gauche	$A \rightarrow A\ \beta \mid \alpha$	$A \rightarrow \alpha\ R_A$ $R_A \rightarrow \beta\ R_A \mid \varepsilon$
Factorisation	$A \rightarrow \alpha\beta \mid \alpha\gamma$	$A \rightarrow \alpha\ N$ $N \rightarrow \beta \mid \gamma$ N étant un nouveau non-terminal.
Substitution (suivie d'une factorisation)	$A \rightarrow B\ \beta \mid G\ \gamma$ $B \rightarrow \alpha\omega_1 \mid \omega_2$ $G \rightarrow \alpha\omega_3 \mid \omega_4$	$A \rightarrow \alpha\omega_1\beta \mid \omega_2\beta \mid \alpha\omega_3\gamma \mid \omega_4\gamma$ $B \rightarrow \alpha\omega_1 \mid \omega_2$ $G \rightarrow \alpha\omega_3 \mid \omega_4$ Puis, factorisation $A \rightarrow \alpha\ N \mid \omega_2\beta \mid \omega_4\gamma$ $N \rightarrow \omega_1\beta \mid \omega_3\gamma$ N étant un nouveau non-terminal.

5.5 Points de génération

5.5.1 Définition et utilisation

Un *point de génération* est un appel de procédure sans paramètre qui est inséré dans le code d'un automate procédural.

Un automate procédural pouvant être automatiquement produit à partir d'une grammaire LL(1), il est possible d'indiquer, dans la grammaire, les endroits des points de génération. On dit que la grammaire est *décorée*.

Pour saisir la manière dont on peut choisir la localisation des points de génération nous traitons un exemple de A-règle décorée (les noms des points de génération sont préfixés par le symbole $ pour les distinguer des terminaux et des non-terminaux) que le lecteur pourra généraliser.

```
$p1 A $p2 →   $p3 a $p4 B $p5 C $p6 b $p7 | $p8 b $p9 A $p10
```

La procédure A est alors:

```
proc A
    p1 ;
    cas Lexlu dans
          qd LexLu apt {a} →
                p3 ;
                AvTete ;
                p4 ;
                B ;
                p5 ;
                C ;
                si LexLu ≠ b alors erreur fsi ;
                p6 ;
                AvTete
                p7 ;
          qd LexLu apt {b} →
          p8 ;
          AvTete ;
          p9 ;
          A ;
          p10 ;
          aut erreur
    fcas ;
    p2
```

On remarque que lors de l'exécution de `p6`, `LexLu` désigne le terminal `b` alors que lors de l'exécution de `p7`, `LexLu` désigne le terminal suivant sur la bande d'entrée.

5.5.2 Exemple

Nous considérons un langage d'expressions comme celui du paragraphe 5.3 et nous souhaitons concevoir un programme qui prend en entrée une telle expression, qui l'analyse et qui produit en sortie une expression équivalente en notation postfixée.

Expression en entrée	Expression en sortie
`(a + b) * (a - b - a)`	`ab+ab-a-*`

Version 1

Dans un premier temps, nous utilisons une grammaire légèrement différente de celle qui est donnée en 5.3, les non-terminaux Opadd et Opmul ayant été supprimés au profit d'une utilisation directe des opérateurs.

$E \rightarrow T\ R_E$
$R_E \rightarrow +\ T\ R_E \mid -\ T\ R_E \mid \varepsilon$
$T \rightarrow F\ R_T$
$R_T \rightarrow *\ F\ R_T \mid :\ F\ R_T \mid \varepsilon$
$F \rightarrow Idf \mid (\ E\ )$
$Idf \rightarrow a \mid b$

Pour ce qui concerne l'automate procédural, la procédure R_E (R_T est construite selon le même schéma) devient:

```
proc R_E
    cas Lexlu dans
        qd LexLu apt {+} →AvTête ; T ; R_E ;
        qd LexLu apt {-} →AvTête ; T ; R_E ;
        qd LexLu apt {), fin} →
        aut erreur
    fcas
```

Pour produire l'expression en sortie, on dispose d'un tampon muni des opérations:

```
produire (unité u)
    -- ajoute l'unité lexicale u en fin du tampon
init
    -- initialise le tampon à vide
vider
    -- affiche le contenu du tampon sur le dispositif de sortie
```

Le travail consiste à décorer la grammaire de points de génération qui peuvent partager des variables globales et à donner les algorithmes des points de génération.

Grammaire décorée

$\$Debut\ S\ \$Fin \rightarrow E$ **fin**
$E \rightarrow T\ R_E$

```
R_E → + T $Plus R_E | - T $Moins R_E | ε
T → F R_T
R_T → * F $Mult R_T | : F $Div R_T | ε
F → Idf | ( E )
Idf → $Una a | $Unb b
```

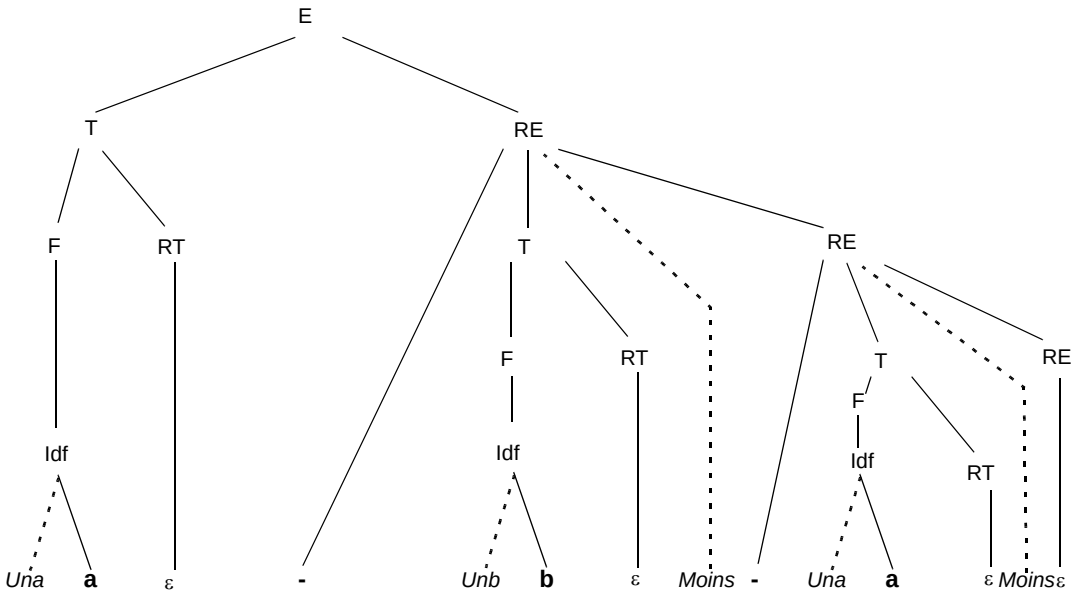
Déclarations

```
Sortie : tampon ;
```

Code des points de génération

```
proc Debut
    Sortie.init

proc Fin
    Sortie.vider

proc Una
    Sortie.produire (a)

proc Unb
    Sortie.produire (b)

proc Plus
    Sortie.produire (+)

proc Moins
    Sortie.produire (-)

proc Mult
    Sortie.produire (*)

proc Div
    Sortie.produire (:)
```

Le schéma ci-dessous décrit l'analyse et la traduction de l'expression a - b - a.



Le point de génération *Moins* suit immédiatement le second opérande de l'opérateur concerné: à l'exécution, la production de l'opérateur est donc faite à la place légitime.

Version 2

Si nous devions admettre d'autres identificateurs que a et b, la multiplication des points de génération comme *Una* et *Unb* serait fastidieuse. Il est clair que lorsque le point de génération *UnIdf*, placé dans la règle F → **$***Unidf* Idf s'exécute, la tête de lecture est positionnée sur l'identificateur concerné. D'où la seconde version.

Grammaire décorée

```
$Debut S $Fin → E fin
E → T RE
RE → + T $Plus RE | - T $Moins RE | ε
T → F RT
RT → * F $Mult RT | : F $Div RT | ε
F → $Unidf Idf | ( E )
Idf → a | b
```

Déclarations

```
Sortie : tampon ;
```

Code des points de génération

```
proc Debut
    Sortie.init

proc Fin
    Sortie.vider
```

```
proc UnIdf
    Sortie.produire (LexLu)

proc Plus
    Sortie.produire (+)

proc Moins
    Sortie.produire (-)

proc Mult
    Sortie.produire (*)

proc Div
    Sortie.produire (:)
```

Dans la méthode d'analyse que nous avons choisi d'exposer, la tête de lecture est le seul moyen de passer des paramètres entre l'analyse syntaxique et l'analyse sémantique. Cette contrainte impose parfois l'usage d'autres techniques pour échanger des informations entre divers moments de la génération. Pour illustrer les difficultés rencontrées, nous cherchons, pour la troisième version, à traiter les opérateurs d'un même niveau de priorité d'une manière homogène.

Version 3

Considérons la règle décorée:

$R_E \rightarrow$ \$*op* Opadd T \$*prod* R_E

Le point de génération *Op* a accès (par le biais de LexLu) à l'opérateur (+ ou -) et peut le mémoriser; le point de génération *Prod* peut récupérer cette information pour produire en sortie l'opérateur concerné.

La question se pose de choisir une structure de mémorisation adéquate. Pour l'analyse de l'expression a - (b - a), l'enchaînement temporel de l'exécution des points de génération est: *UnIdf; Op; UnIdf; Op; UnIdf; Prod; Prod*. Le premier appel de *Op* doit effectuer une mémorisation consommée par le dernier appel de *Prod*, ce qui milite pour une mémorisation en pile.

Nous utilisons donc une pile d'unités lexicales munie des opérations:

```
empiler (unité u)
    -- empile l'unité u s'il reste de la place dans la pile
dépiler : unité
    -- dépile et délivre l'unité en sommet de pile si celle-ci
n'est pas vide
init
    -- initialise une pile vide
```

Grammaire décorée

\$*Debut* S \$*Fin* $\rightarrow$ E **fin**
E $\rightarrow$ T R_E
$R_E \rightarrow$ \$*Op* Opadd T \$*Prod* R_E | ε
T $\rightarrow$ F R_T
$R_T \rightarrow$ \$*Op* Opmul F \$*Prod* R_T | ε

```
F → $Unidf Idf | ( E )
Idf → a | b
Opadd → + | -
Opadd → * | :
```

Déclarations

```
Sortie : tampon ;
Opérateurs:pile ;
```

Code des points de génération

```
proc Debut
    Sortie.init ; Opérateurs.init

proc Fin
    Sortie.vider

proc UnIdf
    Sortie.produire (LexLu)

proc Op
    Opérateurs.empiler (LexLu)

proc Prod
    Sortie.produire (Opérateurs.dépiler)
```

CHAPITRE 6

Évaluation d'expressions arithmétiques et logiques

6.1 Évaluation post-fixée et machine à pile

La notion de pile est étroitement liée aux grammaires algébriques. Ainsi, c'est la pile de récursivité qui donne aux automates procéduraux les capacités de mémorisation nécessaires à l'analyse des langages algébriques.

L'évaluation des expressions arithmétiques peut se faire en utilisant une pile. C'est un schéma d'exécution particulièrement utile.

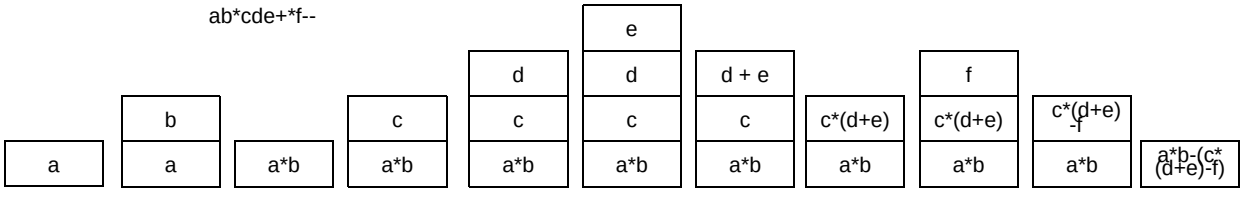
6.1.1 Expressions infixées, postfixées et préfixées

La notation usuelle pour les expressions arithmétiques (dite notation infixe) n'est pas la seule.

- `Notation infixe: a*b - (c * (d + e) - f)`
- `Notation postfixe:ab*cde+*f--`
- `Notation préfixe:-*ab-*c+def`

6.1.2 Évaluation postfixée

La notation postfixe se prête à une évaluation en pile: on empile les opérandes, les opérateurs combinent les deux valeurs en sommet de pile et les remplacent par le résultat.



6.2 La machine BigMach

6.2.1 Schéma général

Dans le cadre de ce cours nous illustrons la partie génération de code en produisant un code intermédiaire (cf. chapitre 1). Le code intermédiaire que nous avons adopté utilise une pile pour effectuer des calculs arithmétiques ou logiques. On peut imaginer une machine, la machine *BigMach*, capable d'exécuter le code intermédiaire (langage machine BigMach) qui nous intéresse. Un simulateur pour cette machine est effectivement disponible sur une plate-forme Unix.

La figure qui suit donne le schéma général de la machine BigMach.

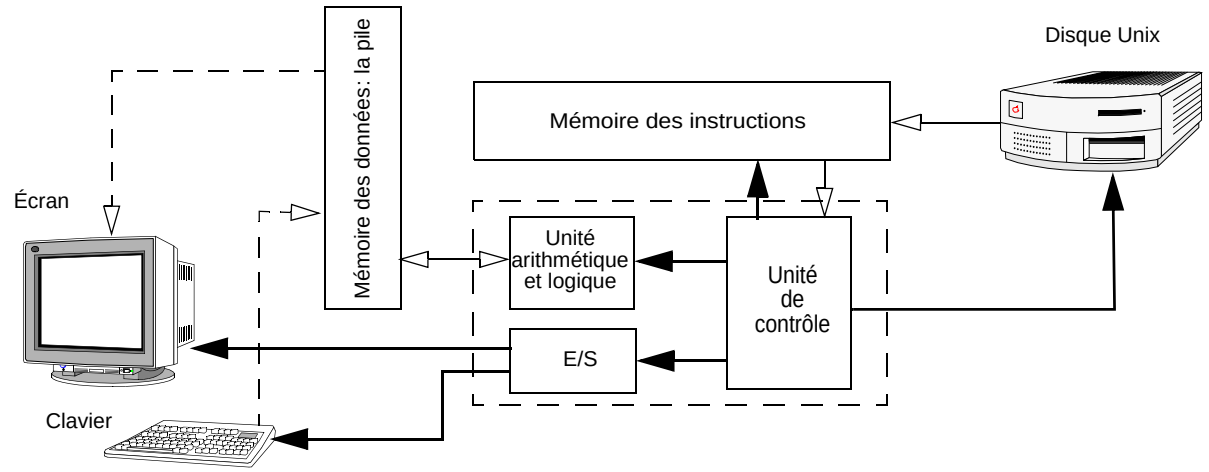


Figure6.1 La machine BigMach

La mémoire des données est constituée d'un tableau adressable de cellules et d'un index repérant le sommet de pile.

```
public static final int taillePile = 1000 ;
private static int [] tPile = new int[taillePile] ;
private static int iPile = -1 ;

public static void empiler (int n) throws BigMachException {
    if (iPile >= taillePile - 1) {throw
              (new BigMachException (BigMachException.
                                     BM_DEBORDEMENT_HAUT)) ; }
    iPile++ ;
    tPile[iPile] = n ;
```

```
}

public static int dépiler () throws BigMachException {
    if (iPile < 0) {throw (new BigMachException (
                    BigMachException.BM_DEBORDEMENT_BAS)) ; }
    iPile-- ;
    return tPile[iPile + 1] ;
}
```

La mémoire des instructions est séparée de la mémoire des données. Il y a deux formats pour les instructions.

- Les instructions au format court occupent un seul mot: il s'agit du code de l'instruction.
- Les instructions au format long occupent deux mots: le code de l'instruction (un mot) et un opérande entier (le second mot).

```
public static final int tailleInsts = 1000 ;
private static int [] tInsts = new int [tailleInsts] ;
```

La mémoire des instructions peut être chargée à partir d'un fichier Unix (format code objet). L'unité de contrôle est munie d'un compteur ordinal `iCo` qui désigne la prochaine instruction à exécuter.

L'unité arithmétique et logique contient deux registres internes.

```
private static int droit ;
private static int gauche ;
```

6.2.2 Répertoire des instructions

Dans un premier temps nous nous limitons au jeu d'instructions suivant :

```
public class CodesInstructions {
    public static final int CODE_STOP           = 0 ;
    public static final int CODE_OU             = 1 ;
    public static final int CODE_ET             = 2 ;
    public static final int CODE_NON            = 3 ;
    public static final int CODE_PLUS           = 4 ;
    public static final int CODE_MOINS          = 5 ;
    public static final int CODE_MULT          = 6 ;
    public static final int CODE_DIV            = 7 ;
    public static final int CODE_DUPL           = 8 ;
    public static final int CODE_INF            = 9 ;
    public static final int CODE_EG             = 10 ;
    public static final int CODE_AFFECTER       = 11 ;
    public static final int CODE_VALEUR         = 12 ;
    public static final int CODE_EMPILER        = 13 ;
    public static final int CODE_DÉPILER        = 14 ;
    public static final int CODE_SOMMETA        = 15 ;
    public static final int CODE_LIRE           = 16 ;
    public static final int CODE_ÉCRIRE         = 17 ;
    public static final int CODE_NOP            = 18 ;
```

```
    public static final int CODE_BSF            = 19 ;
    public static final int CODE_BSV            = 20 ;
    public static final int CODE_TSF            = 21 ;
    public static final int CODE_TSV            = 22 ;
    public static final int CODE_BR             = 23 ;
}
```

6.2.2.1 Instructions arithmétiques et logiques

On dispose d'un certain nombre d'opérations binaires:

- opérations arithmétiques: `PLUS`, `MOINS`, `MULT`, `DIV`;
- opérations de comparaison: `INF`, `EG`;
- opérations logiques: `OU`, `ET`

Les valeurs booléennes sont codées par 0 (`faux`) et 1 (`vrai`); on dispose d'une opération unaire, la négation (`NON`).

L'exécution d'une opération *op* respecte l'algorithme suivant:

Opération binaire

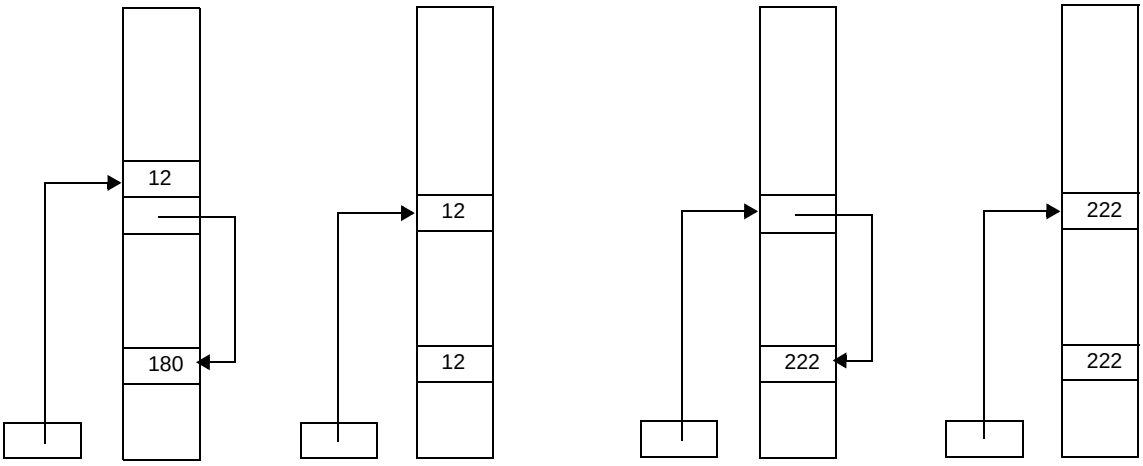
```
droit = PileDeCalcul.dépiler() ;
gauche = PileDeCalcul.dépiler() ;
PileDeCalcul.empiler(gauche op droit);
```

Opération unaire

```
droit = PileDeCalcul.dépiler() ;
PileDeCalcul.empiler(op droit);
```

6.2.2.2 Transferts mémoire/sommet de pile

Les valeurs contenues dans la pile peuvent être des adresses et deux instructions permettent des échanges entre le sommet de pile et un autre emplacement mémoire.



Instruction AFFECT **Instruction** VALEUR

6.2.2.3 Gestion directe de la pile

Trois instructions permettent de gérer explicitement le sommet de pile:

```
EMPILER (val)  -- empile la valeur val
DÉPILER        -- ipile := ipile - 1
SOMMETA(ad)    -- ipile := ad
```

Une instruction, `DUPL`, permet de « dupliquer » la valeur en sommet de pile. Son effet est le suivant:

```
droit = PileDeCalcul.dépiler() ;
PileDeCalcul.empiler(droit);
PileDeCalcul.empiler(droit);
```

6.2.2.4 Branchements

On dispose d'un branchement inconditionnel, `BR (adinst)`, et de quatre branchements conditionnels, `BSV (adinst)` (Branchement si vrai), `TSV (adinst)`, `BSF (adinst)` et `TSF (adinst)`. Ces quatre branchements testent la valeur booléenne placée au sommet de la pile pour décider d'un éventuel branchement: `BSV` et `BSF` dépilent le booléen alors que `TSV` et `TSF` le conservent lorsque le branchement a lieu.

6.2.2.5 Entrées/sorties

Les deux instructions `LIRE (message)` et `ÉCRIRE (message)` réalisent un transfert entre le périphérique (clavier ou écran) et le sommet de pile. Préala-

blement à cet échange, le message (codé) est affiché à l'écran. Dans les deux cas, la valeur transférée est accessible au sommet de pile après l'exécution de l'instruction.

6.3 Le langage BigLang

6.3.1 Syntaxe : une grammaire LL(1) pour BigLang

Un programme BigLang comporte une partie déclarative (où l'on peut déclarer des variables entières ou booléennes) et une partie opérative. À l'exception des instructions d'entrées/sorties, une instruction est une expression combinant, à l'aide d'opérateurs et de parenthèses, des variables, des constantes entières ou des conditionnelles.

Par ordre de priorité croissante, les opérateurs sont :

1. `:=` (seul opérateur binaire associatif à droite)
2. **ou**
3. **et**
4. `<` et `=`
5. `+` et `-`
6. `*` et `/`
7. **non** (opérateur unaire)

Grammaire de BigLang

```
<Prog>::=
    <Decls> debut <Insts> fin
<Decls> ::=
    <Decl> [; <Decls> | vide] | vide
<Decl> ::=
    entier idf <Idents> |
    booleen idf <Idents>
<Idents>::=
    , idf <Idents> | vide
<Insts> ::=
    <Instruction> [ ; <Insts> | vide ]
<Instruction> ::=
    <Exp> | lire idf | ecrire idf
<Exp> ::=
    <SansAff> [ := <Exp> | vide ]
<SansAff> ::=
    <Disjonction>  <RDisjonction>
<RDisjonction> ::=
    ou <Disjonction>  <RDisjonction> | vide
<Disjonction> ::=
    <Conjonction>  <RConjonction>
<RConjonction> ::=
```

```
    et <Conjonction> <RConjonction> | vide
<Conjonction> ::=
    <Relation>  <RRelation>
<RRelation> ::=
    < <Relation> <RRelation> |
    = <Relation> <RRelation> | vide
<Relation> ::=
    <Terme>  <RTerme>
<RTerme> ::=
    + <Terme> <RTerme> |
    - <Terme> <RTerme> | vide
<Terme> ::=
    <Facteur>  <RFacteur>
<RFacteur> ::=
    * <Facteur> <RFacteur> |
    / <Facteur> <RFacteur> | vide
<Facteur> ::=
    non <Facteur> | <Primaire>
<Primaire> ::=
    idf | nombre | ( <Exp> ) |
    si <Exp> alors <Insts> [ sinon <Insts> | vide ] fsi
```

Exemple d'expression

```
i := si c alors j sinon k fsi := 1 < 2 ou 3 + 4 = g
```

6.3.2 Sémantique

6.3.2.1 Biglang est un langage d'expressions

L'expression est donc la construction de base du langage et *toute expression a une valeur, un type (*`entier`*,* `booléen` *ou* `neutre`*), et un attribut (*`rep` *ou* `val`*) indiquant s'il s'agit d'une variable ou non.*

Par abus de langage, nous utilisons parfois le terme type pour désigner à la fois, et le type au sens strict, et l'attribut. Ainsi, le type (au sens étendu) d'une expression pourra être `rep entier` (attribut variable, type `entier`) ou `val entier` (attribut non variable, type `entier`).

6.3.2.2 Règles de typage et d'évaluation des expressions

D'une manière générale, le type `neutre` est le type de l'*absence de valeur* et l'attribut, pour le type `neutre`, ne signifie rien. Pour les types `entier` et `booléen`, les valeurs qui ont l'attribut `val` sont des constantes alors que celles qui ont l'attribut `rep` sont des objets, les *repères*, qui ont la propriété de désigner des valeurs entières ou booléennes. On peut voir les repères comme des adresses. Une valeur du type `rep X` peut être transformée en une valeur du type `val X` par une opération appelée *dérepérage.*

Les règles de typage et d'évaluation des expressions sont définies par induction.

Expressions élémentaires

- Le type d'un identificateur `I` déclaré comme étant de type `X` est `rep X`; sa valeur est un repère `R(I)` attribué statiquement et différent de tout autre repère associé à une autre déclaration.
- Le type d'un nombre `N` est `val entier`; sa valeur est celle de l'entier dont la représentation est `N`.

Expressions arithmétiques et logiques

Nous donnons des règles d'évaluation des expressions arithmétiques et logiques en nous limitant à des opérateurs représentatifs de leur catégorie. Le lecteur généralisera sans peine aux autres opérateurs. Les notations `g` et `d` désignent respectivement les valeurs des opérandes gauches et droits et la notation `derep(r)` le résultat du dérepérage de `r`.

Pour les opérateurs traités dans le tableau ci-dessous, les cas non cités sont des cas d'erreur. Ainsi, l'opérateur `+` ne peut avoir d'opérande du type `neutre`; l'opérateur `:=` ne peut avoir un opérande gauche ayant l'attribut `val`.

	Types			Valeur du résultat[a]
	Gauche	Droit	Résultat	
ou	val booléen	val booléen	val booléen	g ∨ d
	val booléen	rep booléen	val booléen	g ∨ derep(d)
	rep booléen	val booléen	val booléen	derep(g) ∨ d
	rep booléen	rep booléen	val booléen	derep(g) ∨ derep(d)
<	val entier	val entier	val booléen	g < d
	val entier	rep entier	val booléen	g < derep(d)
	rep entier	val entier	val booléen	derep(g) < d
	rep entier	rep entier	val booléen	derep(g) < derep(d)

	Types			Valeur du résultat[a]
	Gauche	Droit	Résultat	
+	val entier	val entier	val entier	g + d
	val entier	rep entier	val entier	g + derep(d)
	rep entier	val entier	val entier	derep(g) + d
	rep entier	rep entier	val entier	derep(g) + derep(d)
:=	rep X	val X	val X	d **Effet de bord**: le repère g désigne la valeur d
	rep X	rep X	val X	derep(d) **Effet de bord**: le repère g désigne la valeur derep(d)

a. Comme il peut y avoir des effets de bords (affectations, E/S), il est important de préciser que toutes les sous-expressions sont évaluées et qu'elles sont évaluées dans l'ordre « naturel ».

Conditionnelle

Nous notons respectivement c, a et s les valeurs des parties condition, alors et sinon. Lorsque la partie sinon est absente, on considère qu'il existe une partie

sinon fictive de type `neutre` dont la valeur est l'absence de valeur (notée `skip`).

Types				Valeur du résultat[a]
Condition	**Alors**	**Sinon**	**Résultat**	
`val booléen`	`val X`	`val X`	`val X`	`a si c, s sinon`
	`val X`	`rep X`	`val X`	`a si c, derep(s) sinon`
	`rep X`	`val X`	`val X`	`derep(a) si c, s sinon`
	`rep X`	`rep X`	`rep X`	`a si c, s sinon`
	`neutre`	`Y`	`neutre`	`skip`
	`Y`	`neutre`	`neutre`	`skip`
	`neutre`	`neutre`	`neutre`	`skip`
`rep booléen`	`val X`	`val X`	`val X`	`a si derep(c), s sinon`
	`val X`	`rep X`	`val X`	`a si derep(c), derep(s) sinon`
	`rep X`	`val X`	`val X`	`derep(a) si derep(c), s sinon`
	`rep X`	`rep X`	`rep X`	`a si derep(c), s sinon`
	`neutre`	`Y`	`neutre`	`skip`
	`Y`	`neutre`	`neutre`	`skip`
	`neutre`	`neutre`	`neutre`	`skip`

a. On commence l'évaluation par la partie condition (qui est toujours évaluée), puis, selon le cas, on poursuit par la partie alors ou la partie sinon.

Séquentialité et instruction vide

Le type de l'instruction vide est `neutre`, sa valeur est `skip`. La valeur d'une séquence est celle de la dernière instruction de la séquence, mais les instructions sont toutes évaluées, dans l'ordre où elles sont écrites. L'effet du point virgule est donc, lorsque le type de l'instruction qui précède n'est pas `neutre`, de *neutraliser* (« d'oublier ») le résultat partiel.

Entrées/sorties

Les instructions d'entrées/sorties sont du type de la variable concernée par l'échange (mais avec l'attribut val) et ont pour valeur la valeur échangée. Elles

ne peuvent être utilisées dans une expression. Elles provoquent des effets de bord:

- échanges mémoire/périphérique;
- affectation de la variable dans le cas d'une lecture.

6.3.3 Schéma d'exécution

Le premier exemple de code objet que nous considérons nous permet d'évo-

Texte source	Code objet	
entier i, j, k ; **booleen** b **debut** lire b ; **lire** i ; j :=**si** i < 50 **alors** k := i := 8*7 + 1 ; **ecrire** i ; k **sinon** 8 + 59 -2 **fsi** ; **ecrire** j **fin**	0 : sommeta 1 : 3 2 : empiler 3 : 3 4 : lire 5 : 1646272544 6 : affect 7 : depiler 8 : empiler 9 : 0 10 : lire 11 : 1763713056 12 : affect 13 : depiler 14 : empiler 15 : 1 16 : empiler 17 : 0 18 : valeur 19 : empiler 20 : 50 21 : inf 22 : Bsf 23 : 50 24 : empiler 25 : 2 26 : empiler 27 : 0 28 : empiler 29 : 8 30 : empiler 31 : 7 32 : mult 33 : empiler 34 : 1	35 : plus 36 : affect 37 : affect 38 : depiler 39 : empiler 40 : 0 41 : valeur 42 : ecrire 43 : 1763713056 44 : depiler 45 : empiler 46 : 2 47 : valeur 48 : Br 49 : 58 50 : empiler 51 : 8 52 : empiler 53 : 59 54 : plus 55 : empiler 56 : 2 57 : moins 58 : affect 59 : depiler 60 : empiler 61 : 1 62 : valeur 63 : ecrire 64 : 1780490272 65 : ecrire 66 : 1718185577 67 : stop

quer les principaux schémas d'évaluation.

Les variables `i`, `j`, `k` et `b` sont implantées aux adresses respectives 0, 1, 2 et 3 de la mémoire des données. En conséquence, la première instruction fixe le sommet de pile à 3.

Expressions arithmétique

La traduction de l'expression `8 + 59 - 2` occupe les mots `50` à `57` de la mémoire des instructions. Il s'agit d'une évaluation post-fixée standard.

Affectations

L'expression `k := i := 8*7 + 1` doit se lire `k := (i := ((8*7) + 1))`; sa traduction occupe les mémoires `24` à `37`. On constate que l'opérateur `:=` est traité comme un autre opérateur (évaluation post-fixée).

Séquentialité

Le point virgule qui suit l'expression `k := i := 8*7 + 1` a sa traduction dans la mémoire `38`: la neutralisation du résultat intermédiaire consiste en un dépilement.

Conditionnelle

Le schéma général de traduction d'une conditionnelle peut être observé entre les mémoires `16` et `57`:

```
16 : -- évaluation de la condition >
…
22 : Bsf      -- branchement si faux à la partie sinon
23 : 50
24 :-- évaluation de la partie alors
…
48 : Br       -- branchement au dessus de la partie sinon
49 : 58
50 : -- évaluation de la partie sinon
…
58 :
```

Sur le second exemple, on observera particulièrement le traitement induit par les règles de typage des conditionnelles.

Texte source	Code objet		

entier i, j, k
;
booleen b ;
debut
lire i ;
k := j := 0 ;
b := i < 50 ;
si b
alors
  j
sinon
  k
fsi := si non
(i < 50)
 alors
si i < 75
  alors
  12
  sinon
  21
fsi
 sinon
i
  fsi ;
si b
alors
  27
fsi ;
ecrire j ;
ecrire k
fin
	0 : sommeta
	1 : 3
	2 : empiler
	3 : 0
	4 : lire
	5 :
	1763713056
	6 : affect
	7 : depiler
	8 : empiler
	9 : 2
	10 : empiler
	11 : 1
	12 : empiler
	13 : 0
	14 : affect
	15 : affect
	16 : depiler
	17 : empiler
	18 : 3
	19 : empiler
	20 : 0
	21 : valeur
	22 : empiler
	23 : 50
	24 : inf
	25 : affect
	26 : depiler
	27 : empiler
	28 : 3
	29 : valeur
		30 : Bsf
		31 : 37
		32 : empiler
		33 : 1
		34 : nop
		35 : Br
		36 : 39
		37 : empiler
		38 : 2
		39 : empiler
		40 : 0
		41 : valeur
		42 : empiler
		43 : 50
		44 : inf
		45 : non
		46 : Bsf
		47 : 66
		48 : empiler
		49 : 0
		50 : valeur
		51 : empiler
		52 : 75
		53 : inf
		54 : Bsf
		55 : 61
		56 : empiler
		57 : 12
		58 : nop
		59 : Br
		60 : 63
			61 : empiler
			62 : 21
			63 : nop
			64 : Br
			65 : 69
			66 : empiler
			67 : 0
			68 : valeur
			69 : affect
			70 : depiler
			71 : empiler
			72 : 3
			73 : valeur
			74 : Bsf
			75 : 79
			76 : empiler
			77 : 27
			78 : depiler
			79 : empiler
			80 : 1
			81 : valeur
			82 : ecrire
			83 :
			1780490272
			84 : depiler
			85 : empiler
			86 : 2
			87 : valeur
			88 : ecrire
			89 :
			1797267488
			90 : ecrire
			91 :
			1718185577
			92 : stop

CHAPITRE 7

Analyse sémantique et génération

7.1 Interface avec l'analyse syntaxique

La tête de lecture de l'analyseur syntaxique est un doublet [catégorie de l'unité lexicale, chaîne des caractères du code source composant l'unité].

```
public class TêteDeLecture {
    public int catégorie ;
    public String unité ;
    public String toString () {
         String enClair = unité ;
         if (enClair.equals("<")) {enClair = "&lt;";}
         return "["+
                   CatégoriesLexicales.enClair[catégorie]+
                   ", "+enClair+"]" ;
    }
}
public static TêteDeLecture tête ;
```

7.1.1 Table des symboles

Le compilateur construit, en analysant les déclarations, une table des symboles qui permet d'associer, à chaque identificateur, des informations utiles pour en contrôler l'usage et pour aider à sa traduction. Dans notre cas la table des symboles permet de trouver, pour chaque identificateur déclaré, son type et son adresse (l'adresse d'implantation prévue lors des diverses exécutions du programme).

Exemple

```
entier i, j, k ;
booléen a, b ;
```

```
entier m ;
```

Identificateur	description	
	type	adr
i	entier	0
j	entier	1
k	entier	2
a	booléen	3
b	booléen	4
m	entier	5

Pour cette gestion, les types et variables utilisés par le module de génération sont:

```
static class Description {
…
    int type ;
    int adr ;
…
}
/**
* table des symboles : les éléments sont des
* <code>Description</code>
*/
public static Hashtable tableDesSymboles ;
/**
* Compteur de réservation des variables. Désigne la prochaine
* adresse libre
* pour implanter une variable.
*/
static int adrVariables ;
```

7.1.2 Production du code objet

Le code objet est produit dans un fichier. On rappelle que le compilateur n'a pas, en général, accès à la mémoire d'exécution du programme.

Pour ce qui nous concerne, l'écriture sur le fichier est réalisée en fin de compilation, le code étant produit dans un tampon; chaque cellule du tampon peut accueillir un code instruction BigMach ou un opérande d'une instruction. Un compteur d'emplacement gouverne le remplissage de ce tampon.

```
/**
* production du code objet
*/
public static CodeObjet objet ;
```

```
/**
* Compteur d'emplacements. désigne le prochain emplacement li-
bre
* dans le code objet.
*/
static int Ce ;
```

Pour modifier le contenu du tampon CodeObjet nous utilisons la méthode:

```
/**
* Modification du <code>i<sup>e</sup></code> mot du tampon. Le
cas échéant,
* modifie <code>iObj</code>
* @param i : l'index du mot à mofifier
* @param v : la nouvelle valeur
* @throws BigLangException : si <code>i></code> n'est pas un
index valide.
*/
public void modifie (int i, int v) throws BigLangException {…}
```

7.1.3 Contrôle de types

Pour effectuer le contrôle de type on utilise deux informations caractérisant l'expression en cours de traitement.

```
/**
* type de la construction en cours de traitement
 */
static int typeCourant ;
/**
* indique si la construction courante est un repère
*/
static boolean cEstUnRepère ;
```

Des informations sur d'autres expressions que l'expression courante peuvent être temporairement conservées. Ainsi, on doit pouvoir disposer d'informations sur les deux opérandes d'un opérateur binaire. Pour des raisons que nous explicitons plus loin, ces informations sont mémorisées dans des piles.

```
public static Stack pileDesOpérateursVus = new Stack () ;
public static Stack pileDesTypesVus = new Stack () ;
public static Stack pileDesRepères = new Stack () ;
```

7.2 Traduction des expressions

7.1.1 Constantes et variables

Les deux `<primaire>`, `idf` et `nbre`, sont les expressions les plus simples du langage *BigLang*.

Règles décorées

```
Primaire -> $var idf | $nbre nbre
```

Points de génération

```
public static void var () throws BigLangException {
    Description d = (Description)
         tableDesSymboles.get(AnalyseSyntaxique.tête.unité) ;
    if (d == null) {
         throw new BigLangException (
                   AnalyseSyntaxique.tête.unité,
                   BigLangException.BL_SEM,
                   BigLangException.BL_SEM_IDF_NON_DECL);
         }
    cEstUnRepère = true ;
    typeCourant = d.type ;

    objet.modifie(Ce, CodesInstructions.CODE_EMPILER) ; Ce++ ;
    objet.modifie(Ce, d.adr); Ce++ ;

}
public static void nbre () throws BigLangException {
    cEstUnRepère = false ;
    typeCourant = TYPE_ENTIER ;

    objet.modifie(Ce, CodesInstructions.CODE_EMPILER) ; Ce++ ;
    objet.modifie(Ce, Integer.parseInt(
                   AnalyseSyntaxique.tête.unité)); Ce++ ;
}
```

7.2.2 Opérateurs binaires et unaires

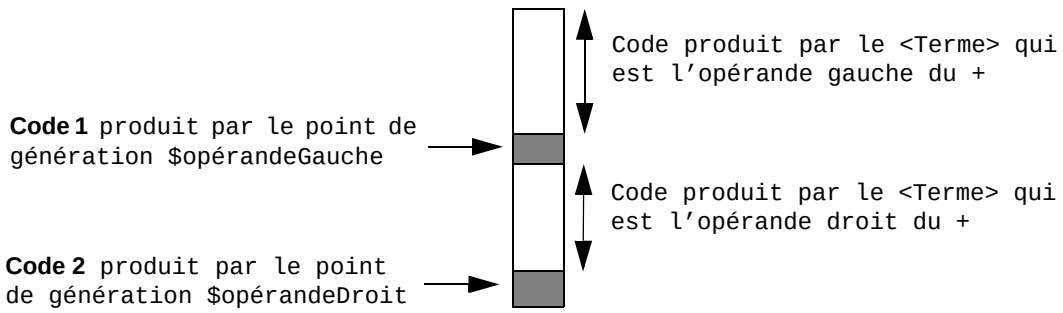
Nous cherchons à produire, pour les opérateurs binaires et unaires un code effectuant une évaluation postfixée et, pour illustrer le cas général, nous considérons le cas de l'opérateur d'addition.

Règles décorées

```
Relation -> Terme RTerme
RTerme -> $opérandeGauche + Terme $opérandeDroit RTerme
```

Schéma de traduction

Nous faisons l'hypothèse que la compilation d'un <Terme> produit, dans le code objet, une séquence de code provoquant son évaluation.



Nous rappelons la sémantique de l'opérateur +:

	Types			Valeur du résultat
	Gauche	Droit	Résultat	
+	val entier	val entier	val entier	g + d
	val entier	rep entier	val entier	g + derep(d)
	rep entier	val entier	val entier	derep(g) + d
	rep entier	rep entier	val entier	derep(g) + derep(d)

Selon cette sémantique, les séquences de code produites respectivement par les points de génération `$opérandeGauche` et `$opérandeDroit` ont pour but:

- d'effectuer un éventuel derepérage du résultat de l'évaluation du premier opérande (Code 1);
- d'effectuer un éventuel derepérage du résultat de l'évaluation du deuxième opérande puis produire une instruction effectuant l'addition (Code 2).

Le contrôle de la conformité des types est faite par un module utilisé pour tous les opérateurs: il est donc nécessaire de disposer de l'intégralité des informations. Le contrôle est donc fait par les points de génération `$opérandeGauche` et `$opérandeGauche`, les caractéristiques de l'opérateur ayant été stockées dans la pile `PileDesOpérateursVus`.

Points de génération

```
public static void opérandeGauche () throws BigLangException {
    Opérateur o = (Opérateur) tableDesOpérateurs.get(
                                AnalyseSyntaxique.tête.unité);
    if (o.typeGauche != TYPE_INEXISTANT
         && o.valeurAGauche
         && cEstUnRepère) {
              objet.modifie(Ce,
                   CodesInstructions.CODE_VALEUR) ; Ce++ ;
              cEstUnRepère = false ;
    }
    if (o.typeGauche != TYPE_INEXISTANT
         && o.typeGauche != typeCourant) {
              throw new BigLangException (
                   o.enClair+": "+typesEnClair[typeCourant],
                   BigLangException.BL_SEM,
                   BigLangException.
                   BL_SEM_TYPE_GAUCHE_NON_CONFORME);};
    pileDesOpérateursVus.push(o) ;
}
public static void opérandeDroit () throws BigLangException {
```

```
Opérateur o = (Opérateur) pileDesOpérateursVus.pop();
if (o.typeDroit != TYPE_INEXISTANT
     && o.valeurADroite && cEstUnRepère) {
          objet.modifie(Ce,
               CodesInstructions.CODE_VALEUR) ; Ce++ ;
          cEstUnRepère = false ;
     }
     if (o.typeDroit != TYPE_INEXISTANT
          && o.typeDroit != typeCourant) {
               throw new BigLangException (
               o.enClair+": "+typesEnClair[typeCourant],
               BigLangException.BL_SEM,
               BigLangException.
                    BL_SEM_TYPE_DROIT_NON_CONFORME);
     }
     objet.modifie(Ce, o.code) ; Ce++ ;
     cEstUnRepère = o.leRésultatEstUnRepère ;
     typeCourant = o.typeRésultat ;
}
```

7.2.3 Conditionnelles

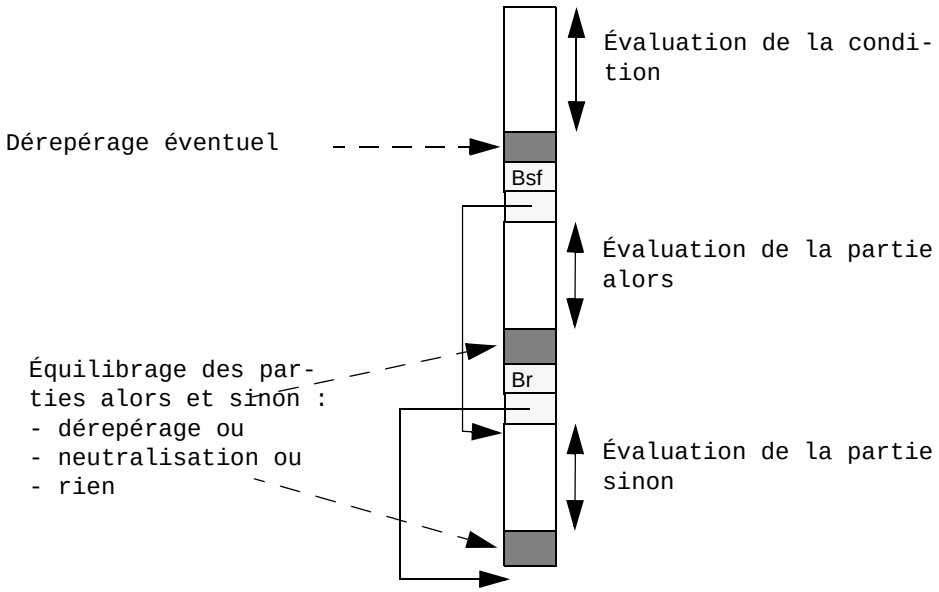
Règles décorées

```
Primaire -> si Exp $finCondition alors Insts RPrimaire fsi
RPrimaire -> $finAlors sinon Insts $finSinon | $fsiSansSinon
vide
```

Schéma de traduction

Deux branchements doivent être implantés: le premier, conditionnel, à la fin de l'évaluation de la condition (il permet de ne pas évaluer la partie « alors »

lorsque la condition s'avère fausse), le second, inconditionnel, à la fin de la partie « alors » (pour éviter d'évaluer la partie « sinon »).



Le code étant produit séquentiellement on ne connaît pas, lorsqu'on génère l'instruction `BSF`, l'adresse concernée par le branchement. On produit alors une valeur arbitraire qui est modifiée lorsque l'information pertinente est disponible (à la fin de la génération de la partie « alors »). L'emplacement, dans le code objet, qui doit accueillir cette information est appelée *point de reprise* et doit être mémorisé. Puisque des conditionnelles peuvent être imbriquée, la structure de mémorisation nécessaire est une pile, la pile des points de reprise.

```
public static Stack pileDesReprises = new Stack () ;
```

Les règles de typage des conditionnelles imposent un équilibrage des parties « alors » et « sinon ». En effet, si l'une des deux parties est une valeur (respectivement de type neutre), l'autre doit être dérepérée (respectivement neutralisée). Cette équilibrage suppose une reprise lorsqu'il doit être effectué à la fin du « alors ». Pour prévoir cette reprise éventuelle nous produisons systématiquement une instruction `NOP` à la fin de la partie « alors », instruction éventuellement remplacée par un dérepérage ou une neutralisation.

Points de génération

```
public static void finCondition () throws BigLangException {
    if (typeCourant != TYPE_BOOLEEN) {
              throw new BigLangException (
                   "si : "+typesEnClair[typeCourant],
                   BigLangException.BL_SEM,
                   BigLangException.
                        BL_SEM_TYPE_BOOL_ATTENDU);
    }
```

```
    if (cEstUnRepère) {
         objet.modifie(Ce,
              CodesInstructions.CODE_VALEUR) ; Ce++ ;
         cEstUnRepère = false ;
    }
    objet.modifie(Ce, CodesInstructions.CODE_BSF) ; Ce++ ;
    pileDesReprises.push(new Integer(Ce));
    objet.modifie(Ce, -1) ; Ce++ ;
}
public static void finAlors () throws BigLangException {
    int adFinCondition = ((Integer)
                          pileDesReprises.pop()).intValue() ;

    objet.modifie(Ce, CodesInstructions.CODE_NOP) ; Ce++ ;
    objet.modifie(Ce, CodesInstructions.CODE_BR) ; Ce++ ;
    pileDesReprises.push(new Integer(Ce));
    objet.modifie(Ce, -1) ; Ce++ ;

    objet.modifie(adFinCondition, Ce) ;

    pileDesTypesVus.push(new Integer(typeCourant)) ;
    pileDesRepères.push(new Boolean(cEstUnRepère)) ;
}
public static void finSinon () throws BigLangException {
    boolean alorsEstUnRep = ((Boolean)
                    pileDesRepères.pop()).booleanValue();
    int typeAlors  = ((Integer)
                    pileDesTypesVus.pop()).intValue() ;
    int adFinAlors = ((Integer)
                    pileDesReprises.pop()).intValue() ;

    if (alorsEstUnRep && !cEstUnRepère) {
         objet.modifie(adFinAlors - 2,
                          CodesInstructions.CODE_VALEUR) ;
    }
    if (!alorsEstUnRep && cEstUnRepère) {
         objet.modifie(Ce,
              CodesInstructions.CODE_VALEUR) ; Ce++ ;
         cEstUnRepère = false ;
    }

    switch (typeAlors) {
    case TYPE_NEUTRE:         // neutraliser le sinon
         if (typeCourant != TYPE_NEUTRE) {
              objet.modifie(Ce,
                   CodesInstructions.CODE_DÉPILER) ; Ce++ ;
              typeCourant = TYPE_NEUTRE ;
              cEstUnRepère = false ;
         }
         break;
    case TYPE_BOOLEEN:
         switch (typeCourant) {
         case TYPE_NEUTRE:    // neutraliser le alors
              objet.modifie(adFinAlors - 2,
```

```
                              CodesInstructions.CODE_DÉPILER) ;
                    break;
               case TYPE_BOOLEEN:
                    break;    // types conformes et équilibrés
               case TYPE_ENTIER:
                    throw new BigLangException (
                         "alors/sinon : "+
                                   typesEnClair[typeCourant],
                         BigLangException.BL_SEM,
                         BigLangException.
                                   BL_SEM_TYPES_NON_CONFORMES);
               default:
                    break;
               }
               break;
          case TYPE_ENTIER:
               switch (typeAlors) {
               case TYPE_NEUTRE:    // neutraliser le alors
                    objet.modifie(adFinAlors - 2,
                              CodesInstructions.CODE_DÉPILER) ;
                    break;
               case TYPE_BOOLEEN:
                    throw new BigLangException (
                         "alors/sinon : "+
                                   typesEnClair[typeCourant],
                         BigLangException.BL_SEM,
                         BigLangException.
                                   BL_SEM_TYPES_NON_CONFORMES);
               case TYPE_ENTIER:
                    break;    // types conformes et équilibrés
               default:
                    break;
               }
               break;
          default:
               break;
          }

          objet.modifie(adFinAlors, Ce) ;
     }
     public static void fsiSansSinon () throws BigLangException {
          if (typeCourant != TYPE_NEUTRE) {
               objet.modifie(Ce,
                         CodesInstructions.CODE_DÉPILER) ; Ce++ ;
               typeCourant = TYPE_NEUTRE ; cEstUnRepère = false ;
          }

          int adFinCondition = ((Integer)
                              pileDesReprises.pop()).intValue() ;
          objet.modifie(adFinCondition, Ce) ;
     }
```

Table des matières

CHAPITRE 1 Introduction à la théorie des langages et à la compilation 5

1.1 Compilation 5
1.1.1 Définitions et terminologie 5
1.1.2 Intérêt 6
1.1.3 Arbres abstraits 7
1.2 Structure d'un compilateur 8
1.2.1 Analyse lexicale 9
1.2.2 Analyse syntaxique 9
1.2.3 Analyse sémantique 9
1.2.4 Génération de code 11
1.2.5 Schéma global 12
1.2.6 Outils pour la construction de compilateurs 12
1.3 Autres membres de la famille 13
1.4 Théorie des langages 13
1.4.1 Vocabulaire et mots 14
1.4.2 Langages 15

CHAPITRE 2 Automates à nombre fini d'états et langages rationnels 17

2.1 Automates déterministes à nombre fini d'états 17
2.1.1 Présentation informelle 17
2.1.2 Définition 18
2.1.3 Automates complets 18
2.1.4 Langage accepté par un automate 19
2.1.5 Notations et variantes terminologiques 20
2.2 Programmation par automates 20
2.2.1 Un problème de programmation 20
2.2.2 Reconnaissance 21
2.2.3 Association d'actions aux transitions 21
2.3 Opérations sur les langages 22
2.3.1 Définitions 22

2.3.2 Exemples 23
2.3.3 Classe des langages rationnels 24
2.3.4 Vers des automates indéterministes 24
2.4 Automates indéterministes 25
2.4.1 Présentation informelle 25
2.4.2 Définitions 25
2.4.3 Déterminisation 26
2.5 Minimisation 27
2.5.1 Automate minimum 27
2.5.2 Algorithme de minimisation 28
2.6 Classe des langages rationnels 30
2.6.1 Inclusion de la classe des rationnels dans la classe des reconnaissables 30
2.6.2 Inclusion de la classe des reconnaissables dans la classe des rationnels 32
2.7 Quelques propriétés des langages rationnels 32
2.7.1 Lemme dit de l'étoile 32
2.7.2 Autres propriétés 33
2.8 Principales applications 33
2.8.1 Analyse lexicale 33
2.8.2 Édition de texte et recherche de motifs 34
2.8.3 Saisie de données 34
2.8.4 Applications parallèles ou réactives 34

CHAPITRE 3 Grammaires et langages 35

3.1 Définition 35
3.1.1 Présentation informelle 35
3.1.2 Vocabulaires 36
3.1.3 Règles 37
3.1.4 Réécriture (dérivation) 37
3.1.5 Résumé 38
3.1.6 Exemples 38
3.2 Classes de langages 40
3.2.1 Classification des grammaires 41
3.2.2 Classes de langages 41
3.2.3 Quelques paradigmes 41
3.3 Appartenance au langage engendré par une grammaire 42
3.3.1 Grammaires de type 1 42
3.3.2 Grammaires de type 0 43
3.3.3 Grammaires de type 3 43
3.4 Grammaire vs automates 45

CHAPITRE 4 Grammaires algébriques 47

4.1 Grammaires algébriques 47
4.1.1 Définition 47
4.1.2 Notations 47
4.1.3 Une grammaire algébrique est une grammaire non contextuelle 48
4.2 Arbres syntaxiques 48
4.2.1 Définition et exemples 48
4.2.2 Propriétés 49

4.2.3 Propriétés 51
4.2.4 Arbres syntaxiques de type 2 et de type 3 52
4.3 Grammaires algébriques et programmes 53
4.3.1 Tirer des dérivations au hasard 53
4.3.2 Grammaires décorées et programmes 54
4.4 Quelques propriétés des grammaires algébriques 55
4.4.1 Élimination des dérivations vides 55
4.4.2 Suppression des non-terminaux superflus 55
4.4.3 Élimination des règles de la forme A → B 55
4.5 Formes normales 55
4.5.1 Forme normale de Chomsky 55
4.5.2 Forme normale de Greibach 55
4.6 Lemme d'itération pour les langages algébriques 56

CHAPITRE 5 Analyse LL(1) 57

5.1 Présentation informelle 57
5.1.1 Analyse LL(1) 57
5.1.2 Automate procédural 58
5.2 Automate procédural 60
5.2.1 Vers une traduction systématique de la grammaire 60
5.2.2 Conditions pour qu'une grammaire soit LL(1) 63
5.3 Une étude de cas: les expressions arithmétiques 64
5.3.1 Grammaire LL(1) pour les expressions arithmétiques 64
5.3.2 Automate procédural 64
5.3.3 Problèmes 65
5.4 Classe des langages LL(1) 66
5.4.1 Principaux résultats 66
5.4.2 Déterminer si une grammaire est LL(1) 67
5.4.3 Heuristiques d'élimination des défauts 69
5.5 Points de génération 70
5.5.1 Définition et utilisation 70
5.5.2 Exemple 71

CHAPITRE 6 Évaluation d'expressions arithmétiques et logiques 77

6.1 Évaluation post-fixée et machine à pile 77
6.1.1 Expressions infixées, postfixées et préfixées 77
6.1.2 Évaluation postfixée 77
6.2 La machine BigMach 78
6.2.1 Schéma général 78
6.2.2 Répertoire des instructions 79
6.3 Le langage BigLang 82
6.3.1 Syntaxe: une grammaire LL(1) pour BigLang 82
6.3.2 Sémantique 83
6.3.3 Schéma d'exécution 87

CHAPITRE 7 Analyse sémantique et génération 91

7.1 Interface avec l'analyse syntaxique 91
7.1.1 Table des symboles 91
7.1.2 Production du code objet 92
7.1.3 Contrôle de types 93
7.2 Traduction des expressions 93
7.2.1 Constantes et variables 93
7.2.2 Opérateurs binaires et unaires 94
7.2.3 Conditionnelles 96