Université de Bretagne Occidentale

IUP Ingénierie INFORMATIQUE
Deuxième année

COMPILATION
THÉORIE DES LANGAGES

Dernière révision : 29 janvier 2003

Table des matières

Chapitre 1

Introduction

1.1 Qu'est ce que la compilation ?

Tout programmeur utilise jour après jour un outil essentiel à la réalisation de programmes informatiques: le **compilateur**. Un compilateur est un logiciel particulier qui traduit un programme écrit dans un langage de haut niveau (par le programmeur) en instructions exécutables (par un ordinateur). C'est donc l'instrument fondamental à la base de tout réalisation informatique.

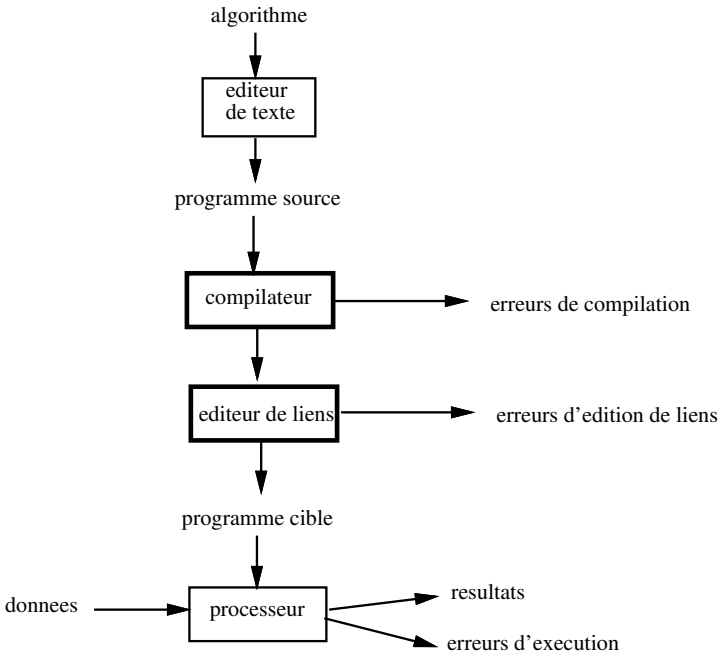


FIG. 1.1 - *Chaîne de développement d'un programme*

Tout programme écrit dans un langage de haut niveau (dans lequel il est fait abstraction (sauf pour quelques instructions) de la structure et des détails du calculateur sur lequel les programmes sont destinés à être exécutés) ne peut être éxécuté par un ordinateur que s'il est **traduit** en instructions exécutables par l'ordinateur (langage machine, instructions élémentaires directement exécutables par le processeur).

Remarque: Une autre phase importante qui intervient après la compilation pour obtenir un programme exécutable est la phase **d'éditions de liens**. Un éditeur de lien résout entre autres les références à des appels de routines dont le code est conservé dans des librairies. En général, un compilateur comprend une partie éditeur de lien. Nous ne parlerons pas de cette étape dans ce cours.
En outre, sur les systèmes modernes, l'édition des liens est faite **à l'exécution** du programme! (le programme est plus petit, et les mises à jour sont plus faciles)
Autre remarque: on ne parlera pas non plus de la précompilation (cf préprocesseur C)

Attention, il ne faut pas confondre les compilateurs et les **interpréteurs**!

• Un compilateur est un **programme** (de traduction automatique d'un programme écrit dans un langage source en un programme écrit dans un langage cible). Le fichier résultant de cette compilation est directement exécu-

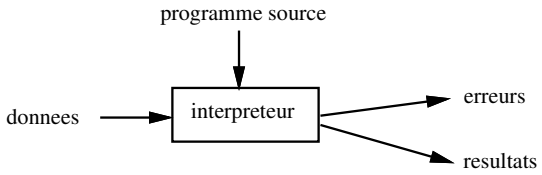


FIG. 1.2 - *Interpréteur*

table une fois pour toutes.
Exemples de langages compilés : Pascal, C, C++, ADA, Fortran, Cobol

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

• Il existe des langages qui sont à mi-chemin de l'interprétation et de la compilation. On les appelle **langages P-code** ou **langages intermédiaires**. Le code source est traduit (compilé) dans une forme binaire compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Lorsque l'on exécute le programme, ce P-code est interprété. Par exemple en Java, le source est compilé pour obtenir un fichier (.class) "byte code" qui sera interprété par une **machine virtuelle**. Autre langage p-code : Python.
Les interpréteurs de p-code peuvent être relativement petits et rapides, si bien que le p-code peut s'exécuter presque aussi rapidement que du binaire compilé[1]. En outre les langages p-code peuvent garder la flexibilité et la puissance des langages interprétés. On ne garde que les avantages !!

1.2 Pourquoi ce cours ?

Il est évident qu'il n'est pas nécessaire de comprendre comment est écrit un compilateur pour savoir comment l'utiliser. De même, un informaticien a peu de chances d'être impliqué dans la réalisation ou même la maintenance d'un compilateur pour un langage de programmation majeur. Alors pourquoi ce cours?

1) La "compilation" n'est pas limitée à la traduction d'un programme informatique écrit dans un langage de haut niveau en un programme directement exécutable par une machine, cela peut aussi être :
• la traduction d'un langage de programmation de haut niveau vers un autre langage de programmation de haut niveau. Par exemple une traduction de Pascal vers C, ou de Java vers C++, ou de ...
Lorsque le langage cible est aussi un langage de haut niveau, on parle plutôt de **traducteur**. Autre différence entre traducteur et compilateur: dans un traducteur, il n'y a pas de perte d'informations (on garde les commentaires, par exemple), alors que dans un compilateur il y a perte d'informations.
• la traduction d'un langage de programmation de bas niveau vers un autre langage de programmation de haut niveau. Par exemple pour retrouver le code C à partir d'un code compilé (piratage, récupération de vieux logiciels, ...)
• la traduction d'un langage **quelconque** vers une autre langage quelconque (ie pas forcemment de programmation) : word vers html, pdf vers ps, ...
Ce genre de travail peut très bien être confié à un ingénieur maître de nos jours.

2) Le but de ce cours est de présenter les principes de base inhérents à la réalisation de compilateurs. Les idées et techniques développées dans ce domaine sont si **générales et fondamentales** qu'un informaticien (et même un scientifique non informaticien) les utilisera très souvent au cours de sa carrière : traitement de données, moteurs de recherche, outils `sed` ou `awk`, etc.
Nous verrons
- les principes de base inhérents à la réalisation de compilateurs: analyse lexicale, analyse syntaxique, analyse sémantique, génération de code,

1. mais *presque* seulement ...

- les outils fondamentaux utilisés pour effectuer ces analyses : fondements de base de la théorie des langages (grammaires, automates, ...), méthodes algorithmiques d'analyse, ...

3) En outre, comprendre comment est écrit un compilateur permet de mieux comprendre les "contraintes" imposées par les différents langages lorsque l'on écrit un programme dans un langage de haut niveau.

1.3 Bibliographie succinte

Ce cours s'inspire des livres suivants

- *A. Aho, R. Sethi, J. Ullman*, **Compilateurs : principes, techniques et outils**, InterEditions 1991.
- *N. Silverio*, **Réaliser un compilateur**, Eyrolles 1995.
- *R.Wilhelm, D. Maurer*, **Les compilateurs : théorie, construction, génération**, Masson 1994.
- *J. Levine, T. Masson, D. Brown*, **lex & yacc**, Éditions O'Reilly International Thomson 1995.

Chapitre 2

Structure d'un compilateur

La compilation se décompose en deux phases :
• une phase d'analyse, qui va reconnaître les variables, les instructions, les opérateurs et élaborer la structure syntaxique du programme ainsi que certaines propriétés sémantiques
• une phase de synthèse et de production qui devra produire le code cible.

2.1 Phases d'analyse

2.1.1 Analyse lexicale

(appelée aussi *Analyse linéaire*)
Il s'agit de reconnaître les "types" des "mots" lus. Pour cela, les caractères sont regroupés en **unités lexicales** (**token**).
L'analyse lexicale se charge de

- éliminer les caractères superflus (commentaires, espaces, passages à la ligne, ...)
- identifier et traiter les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur
- identifier les symboles qui représentent des identificateurs, des constantes réelles, entières, chaînes de caractères, des opérateurs (affectation, addition, ...), des séparateurs (parenthèses, points virgules, ...) [1], les mots clefs du langage, ... C'est cela que l'on appelle des **unités lexicales**.

Par exemple, à partir du morceau de C suivant :

```
if (i<a+b)   // super test
         x=2*x;
```

l'analyseur lexical determinera la suite de token : `mot_clé séparateur ident op_rel ident op_arithm ident séparateur ident affectation cste op_arithm` ...

2.1.2 Analyse syntaxique

(appelée aussi *Analyse hiérarchique* ou *Analyse grammaticale*)
Il s'agit de vérifier que les unités lexicales sont dans le bon ordre défini par le langage ie : regrouper les unités lexicales en structures grammaticales, de découvrir la structure du programme. L'analyseur syntaxique sait comment doivent être construites les expressions, les instructions, les déclarations de variables, les appels de fonctions, ...

Exemple. En C, une sélection simple doit se présenter sous la forme :
`if` (*expression*) *instruction*
Si l'analyseur syntaxique reçoit la suite d'unités lexicales
`MC_IF IDENT OPREL ENTIER` ...
il doit signaler que ce n'est pas correct car il n'y a pas de `(` juste après le `if`

L'analyseur syntaxique produit un arbre syntaxique

2.1.3 Analyse sémantique

(appelée aussi *analyse contextuelle*)

1. Certains séparateurs (ou tous ...) peuvent faire partie des caractères à ignorer

Dans cette phase, on opère certains contrôles (contrôles de type, par exemple) afin de vérifier que l'assemblage des constituants du programme a un sens. On ne peut pas, par exemple, additionner un réel avec une chaîne de caractères, ou affecter une variable à un nombre, ...

2.2 Phases de production

2.2.1 Génération de code

Il s'agit de produire les instructions en langage cible.
En règle générale, le programmeur dispose d'un calculateur concret (cartes équipées de processeurs, puces de mémoire, ...). Le langage cible est dans ce cas défini par le type de processeur utilisé.
Mais si l'on écrit un compilateur pour un processeur donné, il n'est alors pas évident de porter ce compilateur (ce programme) sur une autre machine cible.
C'est pourquoi (entre autres raisons), on introduit des machines dites **abstraites** qui font abstraction des architectures réelles existantes. Ainsi, on s'attache plus aux principes de traduction, aux concepts des langages, qu'à l'architecture des machines.

En général, on produira dans un premier temps des instructions pour une machine abstraite (virtuelle). Puis ensuite on fera la traduction de ces instructions en des instructions directement exécutables par la machine réelle sur laquelle on veut que le programme s'exécute. Ainsi, le portage du code compilé sera facilité, car la traduction en code cible virtuel sera faite une fois pour toutes, indépendamment de la machine cible réelle. Il ne reste plus ensuite qu'à étudier les problèmes spécifiques à la machine cible, et non plus les problèmes de reconnaissance du programme (cf Java).

2.2.2 Optimisation de code

Il s'agit d'améliorer le code produit afin que le programme résultant soit plus rapide.
Il y a des optimisations qui ne dépendent pas de la machine cible : élimination de calculs inutiles (faits en double), propagation des constantes, extraction des boucles des invariants de boucle, ...
Et il y a des optimisations qui dépendent de la machine cible : remplacer des instructions générales par des instructions plus efficaces et plus adaptées, utilisation optimale des registres, ...

2.3 Phases parallèles

2.3.1 Gestion de la table des symboles

La table des symboles est la structure de données utilisée servant à stocker les informations qui concernent les identificateurs du programme source (par exemple leur type, leur emplacement mémoire, leur portée, visibilité, nombre et type et mode de passage des paramètres d'une fonction, ...)
Le remplissage de cette table (la collecte des informations) a lieu lors des phases d'analyse. Les informations contenues dans la table des symboles sont nécessaires lors des analyses syntaxique et sémantique, ainsi que lors de la génération de code.

2.3.2 Gestion des erreurs

Chaque phase peut rencontrer des erreurs. Il s'agit de les détecter et d'informer l'utilisateur le plus précisemment possible : erreur de syntaxe, erreur de sémantique, erreur système, erreur interne. Un compilateur qui se contente d'afficher `syntax error` n'apporte pas beaucoup d'aide lors de la mise au point.
Après avoir détecté une erreur, il s'agit ensuite de la traiter de telle manière que la compilation puisse continuer et que d'autres erreurs puissent être détectées. Un compilateur qui s'arrête à la première erreur n'est pas non plus très performant.
Bien sûr, il y a des limites à ne pas dépasser et certaines erreurs (ou un trop grand nombre d'erreurs) peuvent entrainer l'arrêt de l'exécution du compilateur.

2.4 Conclusion

Au début de la vie du monde (dans les années 50), les compilateurs étaient réputés comme étant des programmes difficiles à écrire. Par exemple, la réalisation du premier compilateur Fortran (1957) a nécessité 18 hommes-années de travail. On a découvert depuis des techniques systèmatiques pour traiter la plupart des tâches importantes

qui sont effectuées lors de la compilation.

Différentes phases de la compilation		Outils théoriques utilisés
Phases d'analyse	analyse lexicale (**scanner**)	expressions régulières automates à états finis
	analyse syntaxique (**parser**)	grammaires automates à pile
	analyse sémantique	traduction dirigée par la syntaxe
Phases de production	génération de code	traduction dirigée par la syntaxe
	optimisation de code	
Gestions parallèles	table des symboles	
	traitement des erreurs	

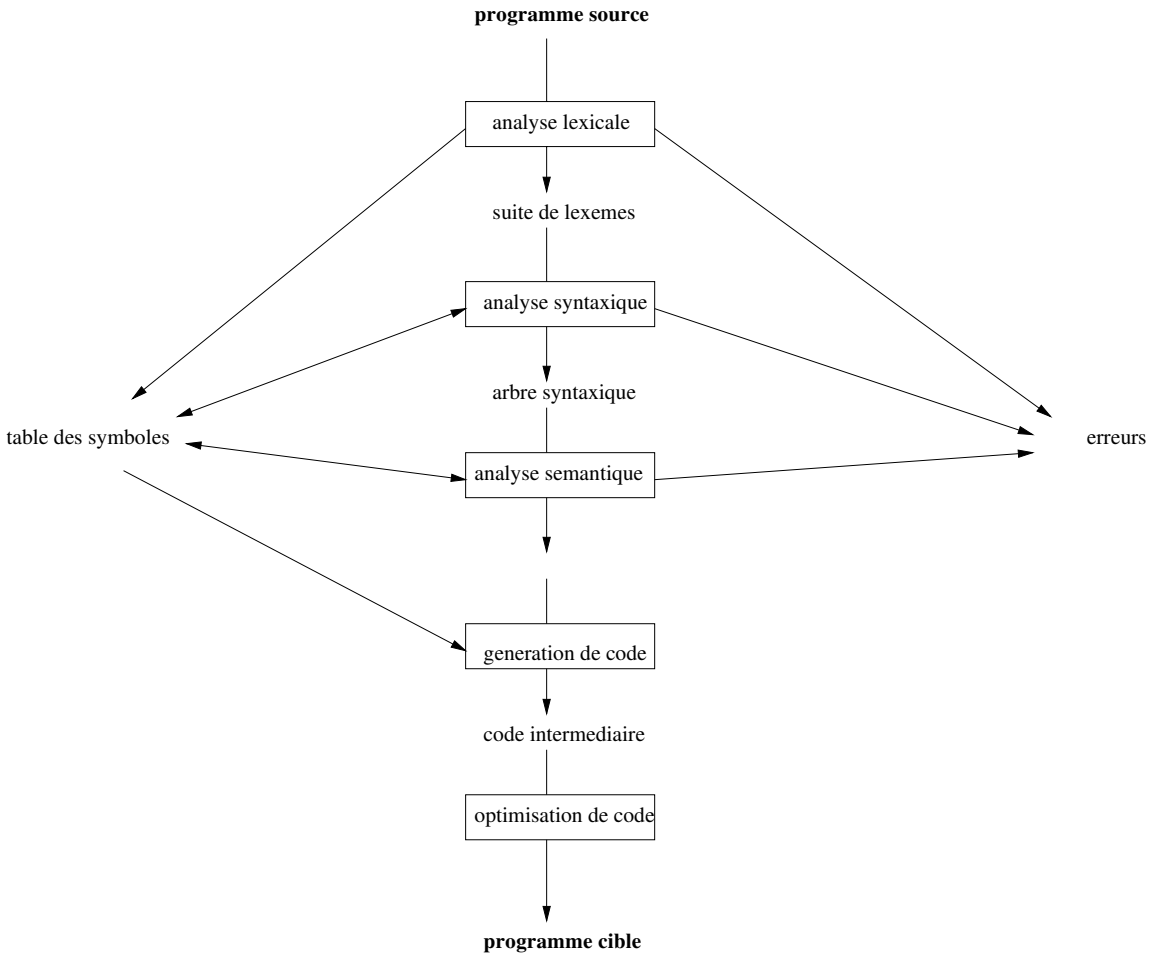


FIG. 2.1 - *Structure d'un compilateur*

Contrairement à une idée souvent répandue, la plupart des compilateurs sont réalisés (écrits) dans un langage de haut niveau, et non en assembleur. Les avantages sont multiples :

- facilité de manipulation de concepts avancés
- maintenabilité accrue du compilateur
- portage sur d'autres machines plus aisé

Par exemple, le compilateur C++ de Björne Stroustrup est écrit en C. Il est même possible d'écrire un compilateur pour un langage L dans ce langage L (`gcc` est écrit en `C`) (bootstrap).

Chapitre 3

Analyse lexicale

L'analyseur lexical constitue la première étape d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'*unités lexicales* que l'analyseur syntaxique aura à traiter. En plus, l'analyseur lexical réalise certaines tâches secondaires comme l'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...), et gère aussi les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient.

Dans ce chapitre, nous abordons des techniques de spécifications et d'implantation d'analyseurs lexicaux.
Ces techniques peuvent être appliquées à d'autres domaines. Le problème qui nous intéresse est la spécification et la conception de programmes qui exécutent des **actions déclenchées par des modèles dans des chaînes** (traitement de données, moteurs de recherche, ...).

3.1 Unités lexicales et lexèmes

Définition 3.1 *Une* **unité lexicale** *est une suite de caractères qui a une signification collective.*

Exemples : les chaînes $\geq, \leq, <, >$ sont des opérateurs relationnels. L'unité lexicale est OPREL (par exemple). Les chaînes `toto`, `ind`, `tab`, `ajouter` sont des identificateurs (de variables, ou de fonctions). Les chaînes `if`, `else`, `while` sont des mots clefs. Les symboles , .; () sont des séparateurs.

Définition 3.2 *Un* **modèle** *est une règle associée à une unité lexicale qui décrit l'ensemble des chaînes du programme qui peuvent correspondre à cette unité lexicale.*

Définition 3.3 *On appelle* **lexème** *toute suite de caractère du programme source qui concorde avec le modèle d'une unité lexicale.*

Exemples :
• L'unité lexicale IDENT (identificateurs) en C a pour modèle : toute suite non vide de caractères composée de chiffres, lettres ou du symbole "_" et qui ne commencent pas par un chiffre. Des exemples de lexèmes pour cette unité lexicale sont : `truc`, `i`,`a1`, `ajouter_valeur` ...
• L'unité lexicale NOMBRE (entier signé) a pour modèle : toute suite non vide de chiffres précédée éventuellement d'un seul caractère parmi $\{+, -\}$. Lexèmes possibles : `-12`, `83204`, `+0` ...
• L'unité lexicale REEL a pour modèle : tout lexème correspondant à l'unité lexicale NOMBRE suivi éventuellement d'un point et d'une suite (vide ou non) de chiffres, le tout suivi éventuellement du caractère `E` ou `e` et d'un lexème correspondant à l'unité lexicale NOMBRE. Cela peut également être un point suivi d'une suite de chiffres, et éventuellement du caractère `E` ou `e` et d'un lexème correspondant à l'unité lexicale NOMBRE. Exemples de lexèmes : `12.4`, `0.5e3`, `10.`, `-4e-1`, `-.103e+2` ...

Pour décrire le modèle d'une unité lexicale, on utilisera des **expressions régulières**.

3.1.1 Expressions régulières

Dans cette partie, nous introduisons quelques notions de base de la **théorie des langages**.

Définition 3.4
On appelle ***alphabet*** *un ensemble fini non vide Σ de symboles (lettres de 1 ou plusieurs caractères).*
On appelle ***mot*** *toute séquence finie d'éléments de Σ.*
On note ε le ***mot vide.***
On note Σ^* l'ensemble infini contenant tous les mots possibles sur Σ.

On note Σ^+ l'ensemble des mots non vides que l'on peut former sur Σ, c'est à dire $\Sigma^+ = \Sigma^* - \{\varepsilon\}$
On note $|m|$ la longueur du mot m, c'est à dire le nombre de symboles de Σ composant le mot.
On note Σ^n l'ensemble des mots de Σ^* de longueur n. Remarque : $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$

Exemples.
• Soit l'alphabet $\Sigma = \{a, b, c\}$. $aaba$, $bbbacbb$, c, ε, ca sont des mots de Σ^*, de longueurs respectives 4, 7, 1, 0 et 2.
• Soit l'alphabet $\Sigma = \{aa, b, c\}$. aba n'est pas un mot de Σ^*. $baab$, caa, bc, $aaaa$ sont des mots de Σ^* de longueurs 3, 2, 2 et 2.

On note . l'opérateur de **concaténation** de deux mots : si $u = u_1 \dots u_n$ (avec $u_i \in \Sigma$) et $v = v_1 \dots v_p$ (avec $v_i \in \Sigma$), alors la concaténation de u et v est le mot $u.v = u_1 \dots u_n v_1 \dots v_p$
Remarque : un mot de n lettres est en fait la concaténation de n mots d'une seule lettre.

Propriété 3.5
- $|u.v| = |u| + |v|$
- $(u.v).w = u.(v.w)$ *(associativité)*
- ε *est l'élément neutre pour* . : $u.\varepsilon = \varepsilon.u = u$

Remarque : nous écrirons désormais uv pour $u.v$

Définition 3.6 *On appelle* **langage** *sur un alphabet Σ tout sous-ensemble de Σ^*.*

Exemples. Soit l'alphabet $\Sigma = \{a, b, c\}$
• Soit L_1 l'ensemble des mots de Σ^* ayant autant de a que de b. L_1 est le langage infini $\{\varepsilon, c, ccc, \dots, ab, ba, \dots, abccc, acbcc, accbc, \dots, aabb, abab, abba, baab, \dots, accbccbccccca, \dots, bbccccaccbcabcccacac, \dots\}$
• Soit L_2 l'ensemble de tous les mots de Σ^* ayant exactement 4 a. L_2 est le langage infini $\{aaaa, aaaac, aaaca, \dots, aabaa, \dots, caaaba, \dots, abcabbbaacc, \dots\}$

Opérations sur les langages :
union : $L_1 \bigcup L_2 = \{w \text{ tq } w \in L_1 \text{ ou } w \in L_2\}$
intersection : $L_1 \bigcap L_2 = \{w \text{ tq } w \in L_1 \text{ et } w \in L_2\}$
concaténation : $L_1 L_2 = \{w = w_1 w_2 \text{ tq } w_1 \in L_1 \text{ et } w_2 \in L_2\}$
$L^n = \{w = w_1 \dots w_n \text{ tq } w_i \in L \text{ pour tout } i \in \{1, \dots, n\}\}$
étoile : $L^* = \cup_{n \geq 0} L^n$

Problème : étant donné un langage, comment décrire tous les mots acceptables ? Comment décrire un langage ?

Il existe plusieurs types de langage (classification) (voir section 7.1), certains étant plus facile à décrire que d'autres. On s'intéresse ici aux **langages réguliers**.

Définition 3.7 *Un* **langage régulier** *L sur un alphabet Σ est défini récursivement de la manière suivante :*
- $\{\varepsilon\}$ *est un langage régulier sur Σ*
- *Si a est une lettre de Σ, $\{a\}$ est un langage régulier sur Σ*
- *Si R est un langage régulier sur Σ, alors R^n et R^* sont des langages réguliers sur Σ*
- *Si R_1 et R_2 sont des langages réguliers sur Σ, alors $R_1 \bigcup R_2$ et $R_1 R_2$ sont des langages réguliers*

Les langages réguliers se décrivent très facilement par une **expression régulière**.

Définition 3.8 *Les* **expressions régulières** *(E.R.) sur un alphabet Σ et les langages qu'elles décrivent sont définis récursivement de la manière suivante :*
- ε *est une E.R. qui décrit le langage $\{\varepsilon\}$*
- *Si $a \in \Sigma$, alors a est une E.R. qui décrit $\{a\}$*
- *Si r est une E.R. qui décrit le langage R, alors $(r)^*$ est une E.R. décrivant R^**
- *Si r est une E.R. qui décrit le langage R, alors $(r)^+$ est une E.R. décrivant R^+*
- *Si r et s sont des E.R. qui décrivent respectivement les langages R et S, alors $(r)|(s)$ est une E.R. décrivant $R \bigcup S$*
- *Si r et s sont des E.R. qui décrivent respectivement les langages R et S, alors $(r)(s)$ est une E.R. dénotant RS*
- *Il n'y a pas d'autres expressions régulières*

Remarques : on conviendra des priorités décroissantes suivantes : $*$, concaténation, $|$ C'est à dire par exemple que $ab^*|c = ((a)((b)^*))|(c)$

En outre, la concaténation est distributive par rapport à $|$: $r(s|t) = rs|rt$ et $(s|t)r = sr|tr$.

Exemples :
• $(a|b)^* = (b|a)^*$ dénote l'ensemble de tous les mots formés de a et de b, ou le mot vide.
• $(a)|((b)^*(c)) = a|b^*c$ est soit le mot a, soit les mots formés de 0 ou plusieurs b suivi d'un c. C'est à dire $\{a, c, bc, bbc, bbbc, bbbbc, \ldots\}$
• $(a^*|b^*)^* = (a|b)^* = ((\varepsilon|a)b^*)^*$
• $(a|b)^*abb(a|b)^*$ dénote l'ensemble des mots sur $\{a, b\}$ ayant le facteur abb
• $b^*ab^*ab^*ab^*$ dénote l'ensemble des mots sur $\{a, b\}$ ayant exactement 3 a
• $(abbc|baba)^+aa(cc|bb)*$

3.2 Mise en œuvre d'un analyseur lexical

3.2.1 Spécification des unités lexicales

Nous disposons donc, avec les E.R., d'un mécanisme de base pour décrire des untiés lexicales. Par exemple, une ER décrivant les identificateurs en C pourrait être :

`ident` = (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v |w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q |R|S|T|U |V|W|X|Y|Z|_) (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r| s|t|u|v|w|x|y|z|A|B |C|D|E|F|G|H|I|J|K|L|M |N| O|P|Q| R|S|T|U|V|W|X|Y|Z|0|1|2|3|4|5|6|7|8|9|_)*

C'est un peu chiant et illisible ...

Alors on s'autorise des **définitions régulières** et le symbole $-$ sur des types ordonnés (lettres, chiffres, ...).
Une définition régulière est une suite de définitions de la forme

$$\begin{cases} d_1 = r_1 \\ \ldots \\ d_n = r_n \end{cases}$$

où chaque r_i est une expression régulière sur l'alphabet $\Sigma \cup \{d_1, \ldots d_{i-1}\}$, et chaque d_i est un nom différent.

Exemple : l'unité lexicale IDENT (identificateurs) en C devient

$$\begin{cases} \text{lettre} = A - Z|a - z \\ \text{chiffre} = 0 - 9 \\ \text{sep} = \_ \\ \text{IDENT} = (\text{lettre} \mid \text{sep})(\text{ lettre} \mid \text{chiffre} \mid \text{sep})^* \end{cases}$$

Remarque IMPORTANTE : toutes les unités lexicales ne peuvent pas être exprimées par des définitions régulières. Par exemple une unité lexicale correspondant au modèle "*toute suite de a et de b de la forme* a^nb^n *avec* $n \geq 0$ " ne peut pas être exprimée par des définitions régulières, car ce n'est pas un langage régulier (mais c'est un langage que l'on appelle *hors contexte*, on verra plus tard qu'on peut s'en sortir avec les langages hors contexte et heureusement).
Autre exemple : il n'est pas possible d'exprimer sous forme d'ER les systèmes de parenthèses bien formés, ie les mots (), (())(), ()(()()(())) .... Ce n'est pas non plus un langage régulier (donc les commentaires à la Pascal ne forment pas un langage régulier).

3.2.2 Attributs

Pour ce qui est des symboles <=, >=, <, >, <>, l'analyseur syntaxique a juste besoin de savoir que cela correspond à l'unité lexicale OPREL (opérateur relationnel). C'est seulement lors de la génération de code que l'on aura besoin de distinguer < de >= (par exemple).
Pour ce qui est des identificateurs, l'analyseur syntaxique a juste besoin de savoir que c'est l'unité lexicale IDENT. Mais le générateur de code, lui, aura besoin de l'adresse de la variable correspondant à cet identificateur. L'analyseur sémantique aura aussi besoin du type de la variable pour vérifier que les expressions sont sémantiquement correctes.
Cette information supplémentaire, inutile pour l'analyseur syntaxique mais utile pour les autres phases du compilateur, est appelée **attribut**.

3.2.3 Analyseur lexical

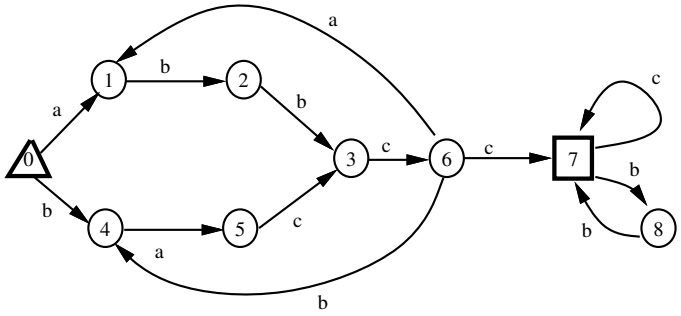
Récapitulons : le rôle d'un analyseur lexical est la reconnaissance des unités lexicales. Une unité lexicale peut (la plupart du temps) être exprimée sous forme de définitions régulières.

Dans la théorie des langages, on définit des **automates** qui sont des machines théoriques permettant la reconnaissance de mots (voir chapitre 7). Ces machines ne marchent que pour certains types de langages. En particulier :

Théorème 3.9 *Un langage régulier est reconnu par un automate fini.*

Contentons nous pour l'instant de donner un exemple d'automate :

Le langage régulier décrit par l'ER $(abbc|bacc)^+c(c|bb)^*$ est reconnu par l'automate :



qui s'écrit également

état	a	b	c
0	1	4	
1		2	
2		3	
3			6
4	5		
5			3
6	1	4	7
7		8	7
8		7	

Bon, un langage régulier (et donc une ER) peut être reconnu par un automate. Donc pour écrire un analyseur lexical de notre programme source, il "suffit" (...) d'écrire un programme simulant l'automate. Lorsqu'une unité lexicale est reconnue, elle envoyée à l'analyseur syntaxique, qui la traite, puis repasse la main à l'analyseur lexical qui lit l'unité lexicale suivante dans le programme source. Et ainsi de suite, jusqu'à tomber sur une erreur ou jusqu'à ce que le programme source soit traité en entier (et alors on est content).

Exemple de morceau d'analyseur lexical pour le langage Pascal (en C) :

```
c = getchar();
switch (c) {
   case ':' : c=getchar();
              if (c== '=')
               {
                 unite_lex = AFFECTATION;
                 c= getchar();
               }
              else
                unite_lex = DEUX_POINTS;
              break;
   case '<' : unite_lex := OPREL;
              c= getchar();
              if (c=='=')
               {
                 attribut = INFEG;
                 c = getchar();
               }
              else
                attribut := INF
              break;
   case .....
```

ou encore, en copiant le travail de l'automate (mais cela donne un code moins rapide, car il contient beaucoup d'appels de fonctions)

```
void Etat1()
```

```
{
  char c;
  c= getchar();
  swicth(c) {
    case ':' : Etat2();
               break;
    case '<' : Etat5();
               break;
    case 'a' : Etat57();
               break;
    ...
    default : ERREUR();
  }
}

void Etat36524()
...
```

Il n'est pas évident du tout dans tout ça de s'y retrouver si on veut modifier l'analyseur, le compléter, ...
Heureusement, il existe des outils pour écrire des programmes simulant des automates à partir de simples définitions régulières. Par exemple : `flex`
Exemple de programme en `flex` :

```
chiffre [0-9]
lettre  [a-zA-Z]
entier  [+-]?[1-9]{chiffre}*
ident   {lettre}({lettre}|{chiffre})*
%%
":="    { return AFF; }
"<="    {
          attribut = INFEG;
          return OPREL;
        }
if|IF|If|iF  {return MC_IF;}
{entier}  {return NB;}
{ident}   {return ID;}
%%
main() {
 yylex();
}
```

3.3 Erreurs lexicales

Peu d'erreurs sont détectables au seul niveau lexical, car un analyseur lexical a une vision très locale du programme source. Les erreurs se produisent lorsque l'analyseur est confronté à une suite de caractères qui ne correspond à aucun des modèles d'unité lexicale qu'il a à sa disposition.

Par exemple, dans un programme C, si l'analyseur rencontre `esle`, s'agit t-il du mot clef `else` mal orthographié ou d'un identificateur? Dans ce cas précis, comme la chaîne correspond au modèle de l'unité lexicale IDENT (identificateur), l'analyseur lexical ne détectera pas d'erreur, et transmettra à l'analyseur syntaxique qu'il a reconnu un IDENT. S'il s'agit du mot clef mal orthographié, c'est l'analyseur syntaxique qui détectera alors une erreur.
Autre exemple : s'il rencontre `1i`, il ne sait pas s'il s'agit du chiffre `1` suivi de l'identificateur `i` ou d'une erreur de frappe (et dans ce cas l'utilisateur pensait-il à `i1`, `i`, `1` ou à tout autre chose?). L'analyseur lexical peut juste signaler que la chaîne `1i` ne correspond à aucune unité lexicale définie.

Lorsque l'analyseur est confronté à une suite de caractères qui ne correspond à aucun de ses modèles, plusieurs stratégies sont possibles :
- **mode panique** : on ignore les caractères qui posent problème et on continue[1]
- transformations du texte source : insérer un caractère, remplacer, échanger, ...

La correction d'erreur par transformations du texte source se fait en calculant le nombre minimum de transformations à apporter au mot qui pose problème pour en obtenir un qui ne pose plus de problèmes. On utilise des

1. attention, il faut tout de même signaler à l'utilisateur que l'on a ignoré ces caractères

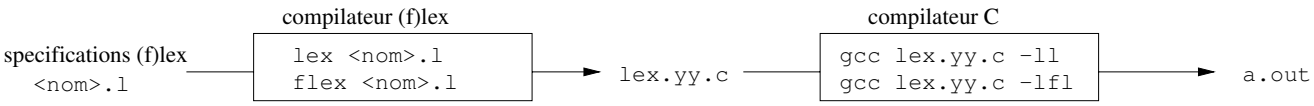
techniques de calcul de distance minimale entre des mots. Cette technique de récupération d'erreur est très peu utilisée en pratique car elle est trop couteuse à implanter. En outre, au niveau lexical, elle est peu efficace, car elle entraine d'autres erreurs lors des phases d'analyse suivantes[2].
La stratégie la plus simple est le mode panique. En fait, en général, cette technique se contente de refiler le problème à l'analyseur syntaxique. Mais c'est efficace puisque c'est lui, la plupart du temps, le plus apte à le résoudre.

2. par exemple, dans le cas du `1i`, le calcul de la distance minimale indiquera qu'il faut considérer `i` tout simplement. Si l'utilisateur pensait en fait à une variable `qi` (par exemple) ça va foutre le bordel. Soit lors de l'analyse sémantique si `i` n'a pas été déclarée, soit lors de l'exécution si `i` sert à autre chose. Le gros problème est alors de savoir gérer la nouvelle erreur

Chapitre 4

L'outil `(f)lex`

De nombreux outils ont été batis pour construire des analyseurs lexicaux à partir de notations spécifiques basés sur des expressions régulières.
`lex` est un utilitaire d'unix. Son grand frère `flex` est un produit gnu. `(f)lex` accepte en entrée des spécifications d'unités lexicales sous forme de définitions régulières et produit un **programme** écrit dans un langage de haut niveau (ici le langage C) qui, une fois compilé, reconnait ces unités lexicales (ce programme est donc un analyseur lexical).



Le fichier de spécifications `(f)lex` contient des expressions régulières suivies d'actions (les **règles de traduction**). L'executable obtenu lit le texte d'entrée caractère par caractère jusqu'à ce qu'il trouve **le plus long** préfixe du texte d'entrée qui corresponde à l'une des expressions régulières. Dans le cas où plusieurs règles sont possibles, c'est la **première** règle rencontrée (de haut en bas) qui l'emporte. Il exécute alors l'action correspondante. Dans le cas où aucune règle ne peut être sélectionnée, l'action **par défaut** consiste à copier le caractère lu en entrée vers la sortie (l'écran).

4.1 Structure du fichier de spécifications `(f)lex`

%{
déclaration (en C) de variables, constantes, ...
%}
déclaration de définitions régulières
%%
règles de traduction
%%
bloc principal et fonctions auxiliaires

• Une définition régulière permet d'associer un nom à une expression régulière `(f)lex` et de se référer par la suite (dans la section des règles) à ce nom plutôt qu'à l'expression régulière.
• les règles de traduction sont des suites d'instructions de la forme

exp_1 action_1
exp_2 action_2
⋮

Les exp_i sont des expressions régulières `(f)lex` et doivent commencer en colonne 0. Les action_i sont des blocs d'instructions en C (i.e. une seule instruction C ou une suite d'instructions entre { et }) qui doivent commencer sur la même ligne que l'expression correspondante.
• la section du bloc principal et des fonctions auxiliaires est facultative (ainsi donc que la ligne **%%** qui la précède). Elles contient les routines C définies par l'utilisateur et éventuellement une fonction `main()` si celle par défaut ne convient pas.

expression	reconnaît, accepte	exemple
c	tout caractère c qui n'est pas un méta-caractère	a
\c	si c n'est pas une lettre minuscule, le caractère c littéralement	\+
"s"	la chaîne de caractère s littéralement	"abc*+"
$r_1 r_2$	r_1 suivie de r_2	ab
.	n'importe quel caractère, excepté le caractère "à la ligne"	a.b
∧	**comme premier caractère** de l'expression, signifie le début de ligne	∧abc
$	**comme dernier caractère** de l'expression, signifie la fin de la ligne	abc$
[s]	n'importe lequel des caractères constituant la chaîne s	[abc] [g-m]
[∧s]	n'importe lequel des caractères à l'exception de ceux constituant la chaîne s	[∧abc]
r^*	0 ou plusieurs occurrences de r	a^*
r^+	1 ou plusieurs occurrences de r	a^+
r?	0 ou 1 occurrence de r	a?
$r\{m\}$	m occurrences de r	a{3}
$r\{m,n\}$	m à n occurrences de r	a{5,8}
$r_1 \mid r_2$	r_1 ou r_2	a|b
r_1 / r_2	r_1 si elle est suivie de r_2	ab/cd
(r)	r	(a|b)?c
\n	caractère "à la ligne"	
\t	tabulation	
{}	pour faire référence à une définition régulière	{nombre}
«EOF»	fin de fichier (UNIQUEMENT avec `flex`)	«EOF»

FIG. 4.1 - *Expressions régulières `(f)lex`*

4.2 Les expressions régulières `(f)lex`

Une expression régulière `(f)lex` se compose de caractères normaux et de méta-caractères qui ont une signification spéciale : $, \, ∧, [,], {, }, <, >, +, −, ∗, /, |,?. La figure 4.1 donne les expressions régulières reconnues par `(f)lex`. Attention, `(f)lex` fait la différence entre les minuscules et les majuscules.

Attention :
- les méta-caractères $, ∧ et / ne peuvent pas apparaître dans des () ni dans des définitions régulières
- le méta caractère ∧ perd sa signification *début de ligne* s'il n'est pas au début de l'expression régulière
- le méta caractère $ perd sa signification *fin de ligne* s'il n'est pas à la fin de l'expression régulière
- à l'intérieur des [], seul \ reste un méta-caractère, le - ne le reste que s'il n'est ni au début ni à la fin.

Priorités :
`truc|machin*` est interprété comme `(truc)|(machi(n*))`
`abc{1,3}` est interprété avec lex comme `(abc){1,3}` et avec flex comme `ab(c{1,3})`
`∧truc|machin` est interprété avec lex comme `(∧truc)|machin` et avec flex comme `∧(truc|machin)`
Avec lex, une référence à une définition régulière n'est pas considérée entre (), en flex si. Par exemple

```
id    [a-z][a-z0-9A-Z]
%%
truc{id}*
```

ne reconnait pas "truc" avec lex, mais le reconnait avec flex.

4.3 Variables et fonctions prédéfinies

char yytext[ ] : tableau de caractères qui contient la chaîne d'entrée qui a été acceptée.
int yyleng : longueur de cette chaîne.
int yylex() : fonction qui lance l'analyseur (et appelle `yywrap()`).
int yywrap() : fonction toujours appelée en fin de flot d'entrée. Elle ne fait rien par défaut, mais l'utilisateur peut la redéfinir dans la section des fonctions auxiliaires. `yywrap()` retourne 0 si l'analyse doit se poursuivre (sur un autre fichier d'entrée) et 1 sinon.

int main() : la fonction `main()` par défaut contient juste un appel à `yylex()`. L'utilisateur peut la redéfinir dans la section des fonctions auxiliaires.
int yylineno : numéro de la ligne courante (ATTENTION : avec lex uniquement).
yyterminate() : fonction qui stoppe l'analyseur (ATTENTION : avec flex uniquement).

4.4 Options de compilation flex

`-d` pour un mode debug
`-i` pour ne pas différencier les majuscules des minuscules
`-l` pour avoir un comportement lex
`-s` pour supprimer l'action par défaut (sortira alors en erreur lors de la rencontre d'un caractère qui ne correspond à aucun motif)

4.5 Exemples de fichier .l

Ce premier exemple reconnait les nombres binaires :

```
%%
(0|1)+              printf("oh un nombre binaire !\n");
```

Tout ce qui n'est pas reconnu comme nombre binaire est affiché à l'écran. Une autre version qui n'affiche QUE les nombres binaires reconnus :

```
%%
(0|1)+              printf("oh le beau  nombre binaire %s !\n", yytext);
.                       // RIEN !!!
```

Ce deuxième exemple supprime les lignes qui commencent par *p* ainsi que tous les entiers, remplace les *o* par des * et retourne le reste inchangé .

```
chiffre    [0-9]
entier     {chiffre}+
%%
{entier}   printf("je censure les entiers");
^p(.)*\n   printf("je deteste les lignes qui commencent par p\n");
o              printf("*");
.          printf("%c",yytext[0]); // action faite par defaut mais c'est mieux de l'expliciter !
```

Ce dernier exemple compte le nombre de voyelles, consonnes et caractères de ponctuations d'un texte entré au clavier.

```
%{
int nbVoyelles, nbConsonnes, nbPonct;
%}
consonne      [b-df-hj-np-xz]
ponctuation   [,;:?!\.]
%%
[aeiouy]        nbVoyelles++;
{consonne}      nbConsonnes++;
{ponctuation}   nbPonct++;
.|\n            // ne rien faire
%%
main(){
  nbVoyelles = nbConsonnes = nbPonct = 0;
  yylex();
  printf("Il y a %d voyelles, %d consonnes et %d signes de ponctuations.\n",
         nbVoyelles, nbConsonnes, nbPonct);
}
```

Chapitre 5

Analyse syntaxique

Tout langage de programmation possède des règles qui indiquent la structure syntaxique d'un programme bien formé. Par exemple, en Pascal, un programme bien formé est composé de blocs, un bloc est formé d'instructions, une instruction de ...
La **syntaxe** d'un langage peut être décrite par une **grammaire**.

L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.
Le problème est donc

$$\left\{ \begin{array}{l} \bullet \text{ étant donnée une grammaire } G \\ \bullet \text{ étant donné un mot } m \text{ (un programme)} \end{array} \right. \Longrightarrow \text{est ce que } m \text{ appartient au langage généré par } G ?$$

Le principe est d'essayer de construire un **arbre de dérivation**. Il existe deux méthodes pour cette construction : méthode (analyse) descendante et méthode (analyse) ascendante.

5.1 Grammaires et Arbres de dérivation

Problème (déja ébauché) : étant donné un langage, comment décrire tous les mots acceptables ? Comment décrire un langage ?
On a déja vu les langages réguliers, qui s'expriment à l'aide d'expressions régulières. Mais la plupart du temps, les langages ne sont pas réguliers et ne peuvent pas s'exprimer sous forme d'une ER. Par exemple, le langage des systèmes de parenthèses bien formés ne peut pas s'exprimer par une ER. On a donc besoin d'un outil plus puissant : les grammaires.

5.1.1 Grammaires

Exemples (définitions informelles de grammaires) :
• dans le langage naturel, une phrase est composée d'un sujet suivi d'un verbe suivi d'un complément (pour simplifier ...).
Par exemple : L'étudiant | subit | un cours
On dira donc :

phrase = *sujet verbe complément*

Ensuite, il faut expliquer ce qu'est un *sujet*, un *verbe*, un *complément*. Par exemple :

sujet = *article adjectif nom*
| *article nom adjectif*
| *article nom*
article = `le` | `la` | `un` | `des` | `l'`
adjectif = `malin` | `stupide` | *couleur*
couleur = `vert` | `rouge` | `jaune`
ainsi de suite ...

• une expression conditionnelle en C est : `if` `(` *expression* `)` *instruction*
Par exemple : `if` `(` $x < 10$ `)` $a = a + b$
Il faut encore définir ce qu'est une *expression* et ce qu'est une *instruction* ...

On distingue les

- *symboles* ***terminaux*** : les lettres du langage (`le`, `la`, `if` ...dans les exemples)
- *symboles* ***non-terminaux*** : les symboles qu'il faut encore définir (ceux en italique dans les exemples

précédents)

Définition 5.1 *Une **grammaire** est la donnée de $G = (V_T, V_N, S, P)$ où*
- *V_T est un ensemble non vide de symboles* **terminaux** *(alphabet terminal)*
- *V_N est un ensemble de symboles* **non-terminaux**, *avec $V_T \bigcap V_N = \emptyset$*
- *S est un symbole initial $\in V_N$ appelé* **axiome**
- *P est un ensemble de* **règles de productions** *(règles de réécritures)*

Définition 5.2 *Une* **règle de production** *$\alpha \rightarrow \beta$ précise que la séquence de symboles α ($\alpha \in (V_T \bigcup V_N)^+$) peut être remplacée par la séquence de symboles β ($\beta \in (V_T \bigcup V_N)^*$).*
α est appellée **partie gauche** *de la production, et β* **partie droite** *de la production.*

Exemple 1 :
symboles terminaux (alphabet) : $V_T = \{a, b\}$
symboles non-terminaux : $V_N = \{S\}$
axiome : S
règles de production : $\begin{cases} S \rightarrow \varepsilon \\ S \rightarrow aSb \end{cases}$ qui se résument en $S \rightarrow \varepsilon \mid aSb$

Exemple 2 : $G =< V_T, V_N, S, P >$ avec
V_T = { il, elle, parle, est, devient, court, reste, sympa, vite }
V_N = { PHRASE, PRONOM, VERBE, COMPLEMENT, VERBETAT, VERBACTION }
S = PHRASE
P = { PHRASE → PRONOM VERBE COMPLEMENT
PRONOM → il | elle
VERBE → VERBETAT | VERBACTION
VERBETAT → est | devient | reste
VERBACTION → parle | court
COMPLEMENT → sympa | vite }

Conventions : l'emploi de lettres capitales est réservé pour dénoter des symboles non-terminaux. Les lettres minuscules du début de l'alphabet sont utilisées pour représenter des symboles terminaux. Les lettres minuscules de la fin de l'alphabet (t, ..., z) sont utilisées pour indiquer une chaîne de symboles terminaux. Les lettres grecques dénotent des chaînes composées de symboles terminaux et non terminaux.

5.1.2 Arbre de dérivation

On appelle **dérivation** l'application d'une ou plusieurs règles à partir d'un mot de $(V_T \bigcup V_N)^+$.
On notera $\rightarrow$ une dérivation obtenue par application d'une seule règle de production, et $\stackrel{*}{\rightarrow}$ une dérivation obtenue par l'application de n règles de production, où $n \geq 0$.

Exemples :
- sur la grammaire de l'exemple 1
  $S \rightarrow \varepsilon$
  $S \rightarrow aSb$
  $aSb \rightarrow aaSbb$
  $S \stackrel{*}{\rightarrow} ab$
  $S \stackrel{*}{\rightarrow} aaabbb$
  $S \stackrel{*}{\rightarrow} aaSbb$
- sur la grammaire de l'exemple 2
  PHRASE → SUJET VERBE COMPLEMENT $\stackrel{*}{\rightarrow}$ elle VERBETAT sympa
  PHRASE $\stackrel{*}{\rightarrow}$ il parle vite
  PHRASE $\stackrel{*}{\rightarrow}$ elle court sympa

Remarque : il est possible de générer des phrases syntaxiquement correctes mais qui n'ont pas de sens. C'est l'analyse sémantique qui permettra d'éliminer ce problème.

Définition 5.3 *Etant donnée une grammaire G, on note $L(G)$ le langage généré par G et défini par*
$$\{w \in (V_T)^* \quad tq\ S \stackrel{*}{\rightarrow} w\}$$
(c'est à dire tous les mots composés uniquement de symboles terminaux (de lettres de l'alphabet) que l'on peut former à partir de S).

L'exemple 1 nous donne $L(G) = \{a^n b^n, n \geq 0\}$

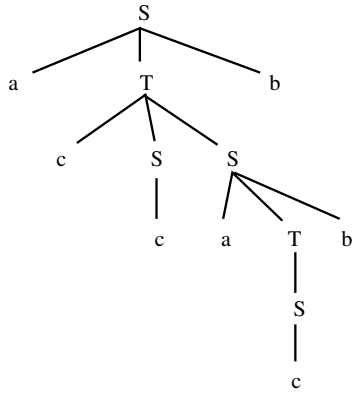
Définition 5.4 *On appelle* **arbre de dérivation** *(ou arbre syntaxique) tout arbre tel que*
- *la racine est l'axiome*

- *les feuilles sont des unités lexicales ou ε*
- *les noeuds sont des symboles non-terminaux*
- *les fils d'un noeud α sont $\beta_0, \ldots, \beta_n$ si et seulement si $\alpha \to \beta_0 \ldots \beta_n$ est une production*

Exemple : Soit la grammaire ayant S pour axiome et pour règles de production

$$P = \begin{cases} S \to aTb \mid c \\ T \to cSS \mid S \end{cases}$$

Un arbre de dérivation pour le mot *accacbb* est :



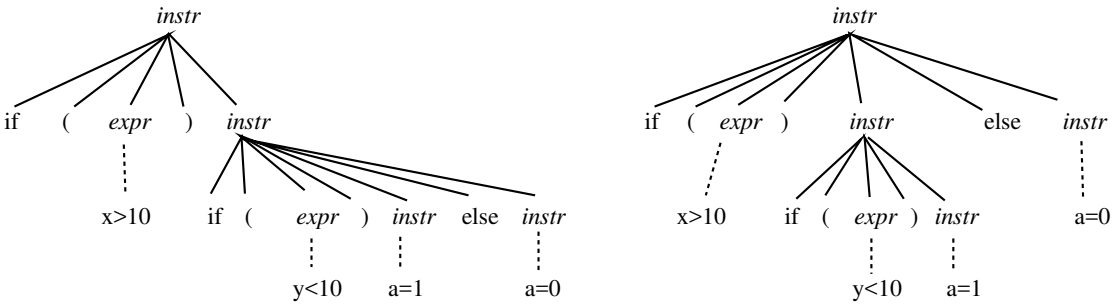
$S \to aTb \to acSSb \to accSb \to accaTbb \to accaSbb \to accacbb$ (dérivations **gauches**)
ou $S \to aTb \to acSSb \to acSaTbb \to acSaSbb \to acSacbb \to accacbb$ (dérivations **droites**)
Ces deux suites **différentes** de dérivations donnent le **même** arbre de dérivation.

Définition 5.5 *Une grammaire est dite* **ambiguë** *s'il existe un mot de $L(G)$ ayant plusieurs arbres syntaxiques.*

Remarque : la grammaire précédente **n'est pas** ambiguë.

Exemple : Soit G donnée par $\begin{cases} instr \to \texttt{if (}expr\texttt{)}\ instr\ \texttt{else}\ instr \\ instr \to \texttt{if (}expr\texttt{)}\ instr \\ instr \to \ldots \\ expr \to \ldots \end{cases}$

Cette grammaire est ambiguë car le mot $m =$ if $(x > 10)$ if $(y < 0)$ $a = 1$ else $a = 0$
possède deux arbres syntaxiques différents :



car il y a deux interprétations syntaxiques possibles :

```
if (x>10)                    if (x>10)
   if (y<0)                       if (y<0)
      a=1                            a=1
   else                           // finsi
      a=0                    else
   // finsi                         a=0
// finsi                     // finsi
```

5.2 Mise en oeuvre d'un analyseur syntaxique

L'analyseur syntaxique reçoit une suite d'unités lexicales (de symboles terminaux) de la part de l'analyseur lexical. Il doit dire si cette suite (ce mot) est syntaxiquement correct, c'est à dire si c'est un mot du langage généré par la grammaire qu'il possède. Il doit donc essayer de construire l'arbre de dérivation de ce mot. S'il y arrive, alors le mot est syntaxiquement correct, sinon il est incorrect.
Il existe deux approches (deux méthodes) pour construire cet arbre de dérivation : une méthode descendante et une méthode ascendante.

5.3 Analyse descendante

Principe : construire l'arbre de dérivation du haut (la racine, c'est à dire l'axiome de départ) vers le bas (les feuilles, c'est à dire les unités lexicales).

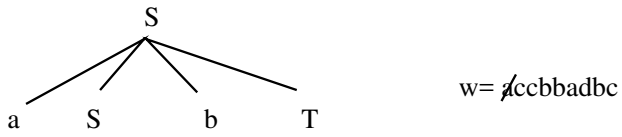
5.3.1 Exemples

• Exemple 1 :

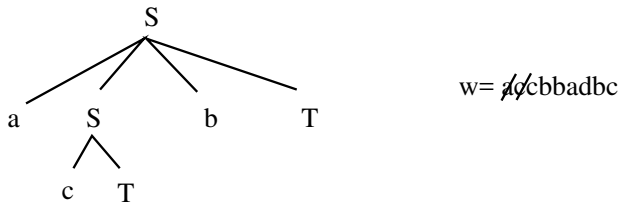
$\begin{cases} S \to aSbT|cT|d \\ T \to aT|bS|c \end{cases}$ avec le mot $w = accbbadbc$

On part avec l'arbre contenant le seul sommet S

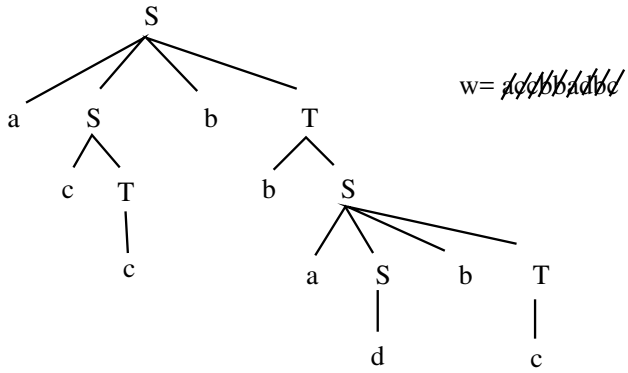
La lecture de la première lettre du mot (a) nous permet d'avancer la construction



puis la deuxième lettre nous amène à



et ainsi de suite jusqu'à



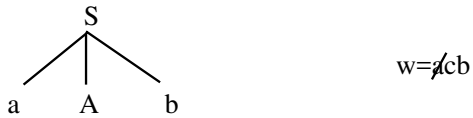
On a trouvé un arbre de dérivation, donc le mot appartient au langage.

Sur cet exemple, c'est très facile parce que chaque règle commence par un terminal différent, donc on sait immédiatement laquelle prendre.

• Exemple 2 :

$\begin{cases} S \to aAb \\ A \to cd|c \end{cases}$ avec le mot $w = acb$

On se retrouve avec



En lisant le c, on ne sait pas s'il faut prendre la règle $A \to cd$ ou la règle $A \to c$. Pour le savoir, il faut lire **aussi** la lettre suivante (b). Ou alors, il faut se donner la possibilité de faire des **retour en arrière :** on essaye la 1ère règle ($A \to cd$), on aboutit à un échec, alors on retourne en arrière et on essaye la deuxième règle et là ça marche.

• Exemple 3 : $S \to aSb|aSc|d$ avec le mot $w = aaaaaaadbbcbbbc$

Pour savoir quelle règle utiliser, il faut cette fois-ci connaître aussi la **dernière** lettre du mot.

• Exemple 4 : grammaire des expressions arithmétiques

$$\begin{cases} E \to TE' \\ E' \to +TE'|-TE'|\varepsilon \\ T \to FT' \\ T' \to *FT'|/FT'|\varepsilon \\ F \to (E)|\text{ nb} \end{cases}$$ avec le mot $w = 3*4+10*(5+11)/34+12$

Pfff ... Alors là on ne voit plus rien du tout !!

Conclusion : ce qui serait pratique, ça serait d'avoir une table qui nous dit : quand je lis tel caractère et que j'en suis à dériver tel symbole non-terminal, alors j'applique telle règle et je ne me pose pas de questions. Ça existe, et ça s'appelle une **table d'analyse**.

5.3.2 Table d'analyse LL(1)

Pour construire une table d'analyse, on a besoin des ensembles PREMIER et SUIVANT

Calcul de PREMIER

Pour toute chaîne α composée de symboles terminaux et non-terminaux, on cherche PREMIER(α) : l'ensemble de tous les **terminaux** qui peuvent **commencer** une chaîne qui se dérive de α
C'est à dire que l'on cherche toutes les lettres a telles qu'il existe une dérivation $\alpha \xrightarrow{*} a\beta$ (β étant une chaîne quelconque composée de symboles terminaux et non-terminaux).

Exemple :

$$\begin{cases} S \to Ba \\ B \to cP|bP|P|\varepsilon \\ P \to dS \end{cases}$$

$S \xrightarrow{*} a$, donc $a \in$ PREMIER(S) $S \xrightarrow{*} cPa$, donc $c \in$ PREMIER(S)
$S \xrightarrow{*} bPa$, donc $b \in$ PREMIER(S) $S \xrightarrow{*} dSa$, donc $d \in$ PREMIER(S)
Il n'y a pas de dérivation $S \xrightarrow{*} \varepsilon$
Donc PREMIER(S)= $\{a, b, c, d\}$
$B \xrightarrow{*} dS$, donc $d \in$ PREMIER(B)
$aB \xrightarrow{*} a\alpha$, donc PREMIER($aB$)= $\{a\}$
$BSb \xrightarrow{*} ab$, donc $a \in$ PREMIER(BS)

Algorithme de construction des ensembles PREMIER :

1. Si X est un non-terminal et $X \to Y_1 Y_2 \ldots Y_n$ est une production de la grammaire (avec Y_i symbole terminal ou non-terminal) alors
   - ajouter les éléments de PREMIER(Y_1) **sauf** ε dans PREMIER(X)
   - s'il existe j ($j \in \{2, \ldots, n\}$) tel que pour tout $i = 1, \ldots, j-1$ on a $\varepsilon \in$PREMIER(Y_i), alors ajouter les éléments de PREMIER(Y_j) **sauf** ε dans PREMIER(X)
   - si pour tout $i = 1, \ldots, n$ $\varepsilon \in$PREMIER(Y_i), alors ajouter ε dans PREMIER(X)
2. Si X est un non-terminal et $X \to \varepsilon$ est une production, ajouter ε dans PREMIER(X)
3. Si X est un terminal, PREMIER(X) = $\{X\}$.

Recommencer jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles PREMIER.

Exemple 1 :

$$\begin{cases} E \to TE' \\ E' \to +TE'|-TE'|\varepsilon \\ T \to FT' \\ T' \to *FT'|/FT'|\varepsilon \\ F \to (E)|\text{ nb} \end{cases}$$

PREMIER(E) = PREMIER(T) = $\{(, \text{nb}\}$
PREMIER(E') = $\{+, -, \varepsilon\}$
PREMIER(T) = PREMIER(F) = $\{(, \text{nb}\}$
PREMIER(T') = $\{*, /, \varepsilon\}$
PREMIER(F) = $\{(, \text{nb}\}$

Exemple 2 :

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

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

Calcul de SUIVANT

Pour tout non-terminal A, on cherche SUIVANT(A) : l'ensemble de tous les symboles terminaux a qui peuvent apparaître immédiatement à droite de A dans une dérivation : $S \xrightarrow{*} \alpha A a \beta$

Exemple :

$$\begin{cases} S \to Sc|Ba \\ B \to BPa|bPb|P|\varepsilon \\ P \to dS \end{cases}$$

$a, b, c, \in$SUIVANT(S) car il y a les dérivations $S \xrightarrow{*} Sc$, $S \xrightarrow{*} dSa$ et $S \xrightarrow{*} bdSba$
$d \in$SUIVANT(B) car $S \xrightarrow{*} BdSaa$
Algorithme de construction des ensembles SUIVANT :

1. Ajouter un marqueur de fin de chaîne (symbole \$ par exemple) à SUIVANT(S)
   (où S est l'axiome de départ de la grammaire)
2. Pour chaque production $A \to \alpha B \beta$ où B est un non-terminal, alors
   ajouter le contenu de PREMIER(β) à SUIVANT(B), **sauf** ε
3. Pour chaque production $A \to \alpha B$, alors ajouter SUIVANT(A) à SUIVANT(B)
4. Pour chaque production $A \to \alpha B \beta$ avec $\varepsilon \in$PREMIER(β), ajouter SUIVANT(A) à SUIVANT(B)

Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT.

Exemple 1 :

$$\begin{cases} E \to TE' \\ E' \to +TE'|-TE'|\varepsilon \\ T \to FT' \\ T' \to *FT'|/FT'|\varepsilon \\ F \to (E)|\text{ nb} \end{cases}$$

SUIVANT(E) $= \{ \$,) \}$
SUIVANT(E') $= \{ \$,) \}$
SUIVANT(T) $= \{ +, -,), \$ \}$
SUIVANT(T') $= \{ +, -,), \$ \}$
SUIVANT(F) $= \{ *, /,), +, -, \$ \}$

Exemple 2 :

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

	PREMIER	SUIVANT
S	a c	\$ b a e
A	a ε	e d
B	b	e d

Construction de la table d'analyse LL

Une table d'analyse est un tableau M à deux dimensions qui indique pour chaque symbole non-terminal A et chaque symbole terminal a ou symbole \$ la règle de production à appliquer.

- Pour chaque production $A \to \alpha$ faire
  1. pour tout $a \in$PREMIER(α) (et $a \neq \varepsilon$), rajouter la production $A \to \alpha$ dans la case $M[A, a]$
  2. si $\varepsilon \in$PREMIER(α), alors pour chaque $b \in$SUIVANT(A) ajouter $A \to \alpha$ dans $M[A, b]$
- Chaque case $M[A, a]$ vide est une erreur de syntaxe

Avec notre exemple (grammaire ETF), on obtient la table figure 5.1

5.3.3 Analyseur syntaxique

Maintenant qu'on a la table, comment l'utiliser pour déterminer si un mot m donné est tel que $S \xrightarrow{*} m$?
On utilise une **pile**.

	nb	+	−	*	/	(	)	$
E	$E \to TE'$					$E \to TE'$		
E'		$E' \to +TE'$	$E' \to -TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$					$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$	$T' \to /FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ nb					$F \to (E)$		

FIG. 5.1 - *Table LL de la grammaire ETF*

Algorithme:

données : mot m terminé par \$, table d'analyse M

initialisation de la pile :

S
\$

et un pointeur ps sur la 1ère lettre de m

repeter
- Soit X le symbole en sommet de pile
- Soit a la lettre pointée par ps
- **Si** X est un non terminal **alors**
  - **Si** $M[X,a] = X \to Y_1 \dots Y_n$ **alors**
    - enlever X de la pile
    - mettre Y_n puis Y_{n-1} puis ...puis Y_1 dans la pile
    - émettre en sortie la production $X \to Y_1 \dots Y_n$
  - **sinon** (case vide dans la table)
    - ERREUR
  - **finsi**
- **Sinon**
  - **Si** $X =$ \$ **alors**
    - **Si** $a =$ \$ **alors** ACCEPTER
    - **Sinon** ERREUR
    - **finsi**
  - **Sinon**
    - **Si** $X = a$ **alors**
      - enlever X de la pile
      - avancer ps
    - **sinon**
      - ERREUR
    - **finsi**
  - **finsi**
- **finsi**

jusqu'à ERREUR ou ACCEPTER\$

Sur l'exemple E, E', T, T', F, soit le mot $m = 3 + 4 * 5$

PILE	Entrée	Sortie
\$ E	$3+4*5$\$	$E \to TE'$
\$ $E'T$	$3+4*5$\$	$T \to FT'$
\$ $E'T'F$	$3+4*5$\$	$F \to$ nb
\$ $E'T'3$	$3+4*5$\$	
\$ $E'T'$	$+4*5$\$	$T' \to \varepsilon$
\$ E'	$+4*5$\$	$E' \to +TE'$
\$ $E'T+$	$+4*5$\$	
\$ $E'T$	$4*5$\$	$T \to FT'$
\$ $E'T'F$	$4*5$\$	$F \to$ nb
\$ $E'T'4$	$4*5$\$	
\$ $E'T'$	$*5$\$	$T' \to *FT'$
\$ $E'T'F*$	$*5$\$	
\$ $E'T'F$	5\$	$F \to$ nb
\$ $E'T'5$	5\$	
\$ $E'T'$	\$	$T' \to \varepsilon$
\$ E'	\$	$E' \to \varepsilon$
\$	\$	ACCEPTER (analyse syntaxique réussie)

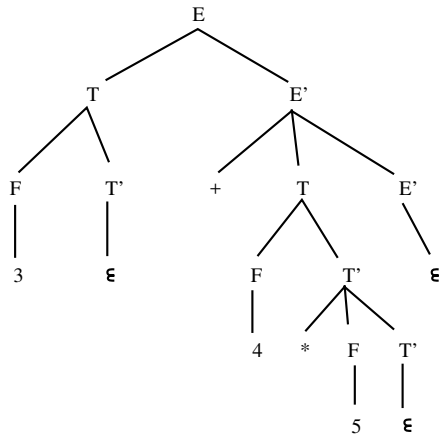


FIG. 5.2 - *Arbre syntaxique pour* $3 + 4 * 5$

On obtient donc l'arbre syntaxique de la figure 5.2.

Si l'on essaye d'analyser maintenant le mot $m = (7 + 3)5$

PILE	Entrée	Sortie
$\$\ E$	$(7+3)5\$$	$E \to TE'$
$\$\ E'T$	$(7+3)5\$$	$T \to FT'$
$\$\ E'T'F$	$(7+3)5\$$	$F \to (E)$
$\$\ E'T')E($	$(7+3)5\$$	
$\$\ E'T')E$	$7+3)5\$$	$E \to TE'$
$\$\ E'T')E'T$	$7+3)5\$$	$T \to FT'$
$\$\ E'T')E'T'F$	$7+3)5\$$	$F \to 7$
$\$\ E'T')E'T'7$	$7+3)5\$$	
$\$\ E'T')E'T'$	$+3)5\$$	$T' \to \varepsilon$
$\$\ E'T')E'$	$+3)5\$$	$E' \to$ * T E'
$\$\ E'T')E'T+$	$+3)5\$$	
$\$\ E'T')E'T$	$3)5\$$	$T \to FT'$
$\$\ E'T')E'T'F$	$3)5\$$	$F \to 3$
$\$\ E'T')E'T'3$	$3)5\$$	
$\$\ E'T')E'T'$	$)5\$$	$T' \to \varepsilon$
$\$\ E'T')E'$	$)5\$$	$E' \to \varepsilon$
$\$\ E'T')$	$)5\$$	
$\$\ E'T'$	$5\$$	**ERREUR !!**

Donc ce mot n'appartient pas au langage généré par cette grammaire.

5.3.4 Grammaire LL(1)

L'algorithme précédent ne peut pas être appliqué à toutes les grammaires. En effet, si la table d'analyse comporte des entrées multiples (plusieurs productions pour une même case $M[A, a]$), on ne pourra pas faire une telle analyse descendante car on ne pourra pas savoir quelle production appliquer.

Définition 5.6 *On appelle* ***grammaire LL(1)*** *une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.*

Le terme "LL(1)" signifie que l'on parcourt l'entrée de gauche à droite (L pour *Left to right scanning*), que l'on utilise les dérivations gauches (L pour *Leftmost derivation*), et qu'**un seul** symbole de pré-vision est nécessaire à chaque étape nécessitant la prise d'une décision d'action d'analyse (1).

Par exemple, nous avons déja vu la grammaire

$$\begin{cases} S \to aAb \\ A \to cd|c \end{cases}$$

Nous avons PREMIER$(S) = \{a\}$, PREMIER$(A) = \{c\}$, SUIVANT$(S) = \{\$\}$ et SUIVANT$(A) = \{b\}$, ce qui donne la table d'analyse

	a	c	b	d	\$
S	$S \to aAb$				
A		$A \to cd$ $A \to c$			

Il y a deux réductions pour la case $M[A, c]$, donc ce n'est pas une grammaire LL(1). On ne peut pas utiliser cette méthode d'analyse. En effet, on l'a déja remarqué, pour pouvoir choisir entre la production $A \to cd$ et la production $A \to c$, il faut lire la lettre qui suit celle que l'on pointe (donc **deux** symboles de pré-vision sont nécessaires) [1].

Autre exemple :

$$\begin{cases} S \to aTbbbb \\ T \to Tb|\varepsilon \end{cases}$$ n'est pas LL(1)

Théorème 5.7 *Une grammaire **ambiguë** ou **récursive à gauche** ou **non factorisée à gauche** n'est pas LL(1)*

"Ambigüe", on a déjà vu, on sait ce que ça veut dire. Voyons les autres termes.

5.3.5 Récursivité à gauche

Définition 5.8 *Une grammaire est **immédiatement récursive à gauche** si elle contient un non-terminal A tel qu'il existe une production $A \to A\alpha$ où α est une chaîne quelconque.*

Exemple :

$$\begin{cases} S \to ScA|B \\ A \to Aa|\varepsilon \\ B \to Bb|d|e \end{cases}$$

Cette grammaire contient plusieurs récursivités à gauche immédiates.

Eliminitation de la récursivité à gauche immédiate:

Remplacer toute règle de la forme $A \to A\alpha|\beta$ par
$A \to \beta A'$
$A' \to \alpha A'|\varepsilon$

Théorème 5.9 *La grammaire ainsi obtenue reconnait le même langage que la grammaire initiale.*

Sur l'exemple, on obtient :

$$\begin{cases} S \to BS' \\ S' \to cAS'|\varepsilon \\ A \to A' \\ A' \to aA'|\varepsilon \\ B \to dB'|eB' \\ B' \to bB'|\varepsilon \end{cases}$$

Cette grammaire reconnait le même langage que la première.
Par exemple, le mot $dbbcaa$ s'obtient à partir de la première grammaire par les dérivations $S \to ScA \to BcA \to BbcA \to BbbcA \to dbbcA \to dbbcAc \to dbbcAaa \to dbbcaa$.
Il s'obtient à partir de la deuxième grammaire par $S \to BS' \to dB'S' \to dbB'S' \to dbbB'S' \to dbbS' \to dbbcAS' \to dbbcA'S' \to dbbcaA'S' \to dbbcaaA'S' \to dbbcaaS' \to dbbcaa$.

Remarque : ici on peut se passer de A'.
C'est à dire en fait que $S \to S\alpha|\varepsilon$ est équivalent à $S \to \alpha S|\varepsilon$ (qui, elle, n'est pas immédiatement récursive à gauche, mais à droite, ce qui n'a rien à voir).

Définition 5.10 *Une grammaire est **récursive à gauche** si elle contient un non-terminal A tel qu'il existe une dérivation $A \overset{+}{\to} A\alpha$ où α est une chaîne quelconque.*

Exemple : $\begin{cases} S \to Aa|b \\ A \to Ac|Sd|c \end{cases}$
Le non-terminal S est récursif à gauche car $S \to Aa \to Sda$ (mais il n'est pas immédiatement récursif à gauche).

1. en fait, c'est une grammaire LL(2)

Eliminitation de la récursivité à gauche pour toute grammaire sans règle $A \to \varepsilon$:

Ordonner les non-terminaux $A_1, A_2, \ldots, A_n$
Pour i=1 à n faire
    pour j=1 à i-1 faire
        remplacer chaque production de la forme $A_i \to A_j\alpha$ où $A_j \to \beta_1|\ldots|\beta_p$ par
            $A_i \to \beta_1\alpha|\ldots|\beta_p\alpha$
    fin pour
    éliminer les récursivités à gauche immédiates des productions A_i
fin pour

Théorème 5.11 *La grammaire ainsi obtenue reconnait le même langage que la grammaire initiale.*

Sur l'exemple :
On ordonne S, A
$i = 1$ pas de récursivité immédiate dans $S \to Aa|b$
$i = 2$ et $j = 1$ on obtient $A \to Ac|Aad|bd|\varepsilon$
    on élimine la rec. immédiate :
        $A \to bdA'|cA'$
        $A' \to cA'|adA'|\varepsilon$
Bref, on a obtenu la grammaire :
$$\begin{cases} S \to Aa|b \\ A \to bdA'|A' \\ A' \to cA'|adA'|\varepsilon \end{cases}$$
Autre exemple :
$$\begin{cases} S \to Sa|TSc|d \\ T \to TbT|\varepsilon \end{cases}$$
On obtient la grammaire :
$$\begin{cases} S \to TScS'|dS' \\ S' \to aS'|\varepsilon \\ T \to T' \\ T' \to bTT'|\varepsilon \end{cases}$$
Or on a $S \to TScS' \to T'ScS' \to ScS'$ damned une récursivité à gauche !!!! Eh oui, l'algo ne marche pas toujours lorsque la grammaire possède une règle $A \to \varepsilon$!

5.3.6 Grammaire propre

Définition 5.12 *Une grammaire est dite* ***propre*** *si elle ne contient aucune production $A \to \varepsilon$*

Comment rendre une grammaire propre ? En rajoutant une production dans laquelle le A est remplacé par ε, ceci pour chaque A apparaîssant en partie droite d'une production, et pour chaque A d'un $A \to \varepsilon$.
Exemple :
$$\begin{cases} S \to aTb|aU \\ T \to bTaTA|\varepsilon \\ U \to aU|b \end{cases} \text{ devient } \begin{cases} S \to aTb|ab|aU \\ T \to bTaTA|baTA|bTaA|baA \\ U \to aU|b \end{cases}$$

5.3.7 Factorisation à gauche

L'idée de base est que pour développer un non-terminal A quand il n'est pas évident de choisir l'alternative à utiliser (ie quelle production prendre), on doit réécrire les productions de A de façon à **différer** la décision jusqu'à ce que suffisamment de texte ait été lu pour faire le bon choix.

Exemple :
$$\begin{cases} S \to bacdAbd|bacdBcca \\ A \to aD \\ B \to cE \\ C \to \ldots \\ E \to \ldots \end{cases}$$
Au départ, pour savoir s'il faut choisir $S \to bacdAbd$ ou $S \to bacdBcca$, il faut avoir lu la 5ième lettre du mot (un a ou un c). On ne peut donc pas dès le départ savoir quelle production prendre. Ce qui est incompatible avec une grammaire LL(1). (Remarque : mais pas avec une grammaire LL(5), mais ce n'est pas notre problème.)

Factorisation à gauche :

Pour chaque non-terminal A
trouver le plus long préfixe α commun à deux de ses alternatives ou plus
Si $\alpha \neq \epsilon$, remplacer $A \to \alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|\ldots|\gamma_p$ (où les γ_i ne commencent pas par α) par
$A \to \alpha A'|\gamma_1|\ldots|\gamma_p$
$A' \to \beta_1|\ldots|\beta_n$
Recommencer jusqu'à ne plus en trouver.

Exemple : $\begin{cases} S \to aEbS|aEbSeB|a \\ E \to bcB|bca \\ B \to ba \end{cases}$

Factorisée à gauche, cette grammaire devient : $\begin{cases} S \to aEbSS'|a \\ S' \to eB|\varepsilon \\ E \to bcE' \\ E' \to B|a \\ B \to ba \end{cases}$

5.3.8 Conclusion

Si notre grammaire est LL(1), l'analyse syntaxique peut se faire par l'analyse descendante vue ci-dessus. Mais comment savoir que notre grammaire est LL(1) ? [2]
Etant donnée une grammaire

1. la rendre non ambiguë.
   Il n'y a pas de méthodes. Une grammaire ambiguë est une grammaire qui a été mal concue.
2. éliminer la récursivité à gauche si nécessaire
3. la factoriser à gauche si nécessaire
4. construire la table d'analyse

Il ne reste plus qu'a espérer que ça soit LL(1). Sinon, il faut concevoir une autre méthode pour l'analyse syntaxique [3].

Exemple : grammaire des expressions arithmétiques avec les opérateurs $+ - /$ et $*$
$E \to E + E|E - E|E * E|E/oE|(E)|\text{nb}$
Mais elle est ambiguë. Pour lever l'ambiguïté, on considère les priorités classiques des opérateurs et on obtient la grammaire non ambiguë :

$$\begin{cases} E \to E + T|E - T|T \\ T \to T * F|T/F|F \\ F \to (E)|\text{nb} \end{cases}$$

Après suppression de la récursivité à gauche, on obtient

$$\begin{cases} E \to TE' & T' \to *FT'|/FT'|\varepsilon \\ E' \to +TE'|-TE'|\varepsilon & F \to (E)|\text{ nb} \\ T \to FT' \end{cases}$$

Inutile de factoriser à gauche.
Cette grammaire est LL(1) (c'est l'exemple que l'on a utilisé tout le temps).

Autre exemple : la grammaire $\begin{cases} S \to aTb|\varepsilon \\ T \to cSa|d \end{cases}$ n'est pas LL(1). Or elle n'est pas récursive à gauche, elle est factorisée à gauche et elle n'est pas ambiguë !

5.4 Analyse ascendante

Principe : construire un arbre de dérivation du bas (les feuilles, ie les unités lexicales) vers le haut (la racine, ie l'axiome de départ).
Le modèle général utilisé est le modèle par **décallages-réductions**. C'est à dire que l'on ne s'autorise que deux opérations :

- **décallage (shift)** : décaller d'une lettre le pointeur sur le mot en entrée
- **réduction (reduce)** : réduire une chaîne (suite consécutive de terminaux et non terminaux à gauche du pointeur sur le mot en entrée et finissant sur ce pointeur) par un non-terminal en utilisant une des règles de production

2. Attention, le théorème 5.7 **ne dit pas** que tout grammaire non ambiguë, non récursive à gauche et factorisée à gauche est LL(1).
3. si la grammaire est LL(k), on peut utiliser une méthode descendante en modifiant légèrement la définition des ensembles PREMIER et SUIVANT (cf exercices)

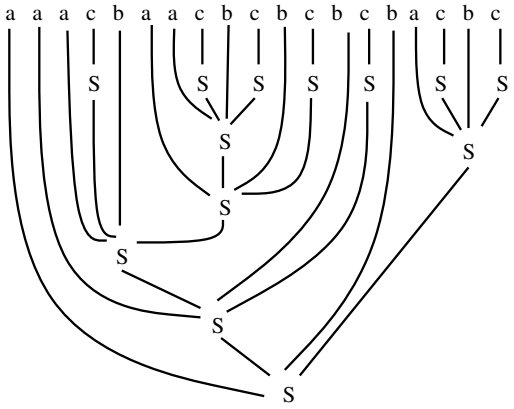


FIG. 5.3 - *Analyse ascendante*

Exemple : $S \to aSbS|c$ avec le mot $u = aacbaacbcbcbacbc$

[a] $acbaacbcbcbacbc$ on ne peut rien **réduire**, donc on **décalle**
a[a] $cbaacbcbcbacbc$ on ne peut rien réduire, donc on décalle
aa[c] $baacbcbcbacbc$ ah ! On peut réduire par $S \to c$
aa[S] $baacbcbcbacbc$ on ne peut rien réduire, donc on décalle
...
$aaSbaa$[c] $bcbcbacbc$ on peut utiliser $S \to c$
$aaSbaa$[S] $bcbcbacbc$ on ne peut rien réduire, donc on décalle
$aaSbaaS$[b] $cbcbacbc$ on ne peut rien réduire, donc on décalle
$aaSbaaSb$[c] $bcbacbc$ On peut utiliser $S \to c$
$aaSbaaSb$[S] $bcbacbc$ On peut utiliser $S \to aSbS$
$aaSba$[S] $bcbacbc$ décallage
$aaSbaS$[b] $cbacbc$ décallage
$aaSbaSb$[c] $bacbc$ réduction par $S \to c$
$aaSbaSb$[S] $bacbc$ réduction par $S \to aSbS$
$aaSb$[S] $bacbc$ réduction par $S \to aSbS$
a[S] $bacbc$ décallage
aS[b] $acbc$ décallage
aSb[a] cbc décallage
$aSba$[c] bc réduction par $S \to c$
$aSba$[S] bc décallage
$aSbaS$[b] c décallage
$aSbaSb$[c] réduction par $S \to c$
$aSbaSb$[S] réduction par $S \to aSbS$
aSb[S] réduction par $S \to aSbS$
[S] **terminé !!!!** On a gagné, le mot est bien dans le langage.

En même temps, on a construit l'arbre (figure 5.3).
Conclusion : ça serait bien d'avoir une table qui nous dit si on décalle ou si on réduit, et par quoi, lorsque le pointeur est sur une lettre donnée.

Table d'analyse SLR
(on l'appelle comme ça parce que, de même que la méthode descendante vue précédemment ne permettait d'analyser que les grammaires LL(1), cette méthode va permettre d'analyser les grammaires dites LR).
Cette table va nous dire ce qu'il faut faire quand on lit une lettre a et qu'on est dans un état i

- soit on **décalle**. Dans ce cas, on empile la lettre lue et on va dans un autre état j. Ce qui sera noté dj
- soit on **réduit** par la règle de production numéro p, c'est à dire qu'on remplace la chaîne en sommet de pile (qui correspond à la partie droite de la règle numéro p) par le non-terminal de la partie gauche de la règle de production, et on va dans l'état j qui dépend du non-terminal en question. On note ça rp
- soit on **accepte** le mot. Ce qui sera noté ACC
- soit c'est une **erreur**. Case vide

Construction de la table d'analyse : utilise aussi les ensembles SUIVANT (et donc PREMIER), plus ce qu'on appelle des fermetures de 0-items. Un 0-item (ou plus simplement *item*) est une production de la grammaire avec un "." quelque part dans la partie droite. Par exemple (sur la gram ETF) : $E \to E.+T$ ou encore $T \to F.$ ou encore $F \to .(E)$

Fermeture d'un ensemble d'items I :

1- Mettre chaque item de I dans Fermeture(I)
2- Pour chaque item i de Fermeture(I) de la forme $A \to \alpha.B\beta$
  pour chaque production $B \to \gamma$
    rajouter (s'il n'y est pas déja) l'item $B \to .\gamma$ dans Fermeture(I)
  finpour
 finpour
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Exemple : soit la grammaire ETF (des expressions arithmétiques)

$$\begin{cases} (1) \quad E \to E+T & (3) \quad T \to T*F & (5) \quad F \to (E) \\ (2) \quad E \to T & (4) \quad T \to F & (6) \quad F \to \text{nb} \end{cases}$$

et soit l'ensemble d'items $\{T \to T*.F\ ,\ E \to E.+T\}$. La fermeture de cet ensemble d'items est :
$\{T \to T*.F\ ,\ E \to E.+T\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\}$

Transition par X d'un ensemble d'items I :

$\Delta(I,X)$ =Fermeture(tous les items $A \to \alpha X.\beta$) où $A \to \alpha.X\beta \in I$

Sur l'exemple ETF :
Soit l'ensemble d'items $I = \{T \to T*.F\ ,\ E \to E.+T\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\}$, on aura
$\Delta(I,F) = \{T \to T*F.\}$
$\Delta(I,+) = \{E \to E+.T\ ,\ T \to .T*F\ ,\ T \to .F\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\}$
etc.

Collection des items d'une grammaire :

0- Rajouter l'axiome S' avec la production : $S' \to S$
1- $I_0 \leftarrow$ Fermeture($\{S' \to .S\}$)
 Mettre I_0 dans Collection
2- Pour chaque $I \in$ Collection
  Pour chaque X tq $\Delta(I,X)$ est non vide
    ajouter $\Delta(I,X)$ dans Collection
  finpour
 finpour
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Toujours sur la grammaire ETF :
$I_0 = \{S \to .E, E \to .E+T, E \to .T, T \to .T*F, T \to .F, F \to .nb, F \to .(E)\}$
$\Delta(I_0,E) = \{S' \to E., E \to E.+T\} = I_1$ (terminal pour la règle $S' \to E$)
$\Delta(I_0,T) = \{E \to T., T \to T.*F\} = I_2$ (terminal pour la règle 2)
$\Delta(I_0,F) = \{T \to F.\} = I_3$ (terminal pour la règle 4)
$\Delta(I_0,() = \{F \to (.E), E \to .E+T, E \to .T, T \to .T*F, T \to .F, F \to .nb, F \to .(E)\} = I_4$
$\Delta(I_0,nb) = \{F \to nb.\} = I_5$ (terminal pour la règle 6)
$\Delta(I_1,+) = \{E \to E+.T, T \to .T*F, T \to .F, F \to .nb, F \to .(E)\} = I_6$
$\Delta(I_2,*) = \{T \to T*.F, F \to .nb, F \to .(E)\} = I_7$
$\Delta(I_4,E) = \{F \to (E.), E \to E.+T\} = I_8$
$\Delta(I_4,T) = \{E \to T., T \to T.*F\} = I_2$ déja vu, ouf
$\Delta(I_4,F) = \{T \to F.\} = I_3$
$\Delta(I_4,nb) = \{F \to nb.\} = I_5$
$\Delta(I_4,() = \{F \to (.E), E \to .E+T, E \to .T, T \to .T*F, T \to .F, F \to .nb, F \to .(E)\} = I_4$
$\Delta(I_6,T) = \{E \to E+T., T \to T.*F\} = I_9$ (terminal pour règle 1)
$\Delta(I_6,F) = I_3$
$\Delta(I_6,nb) = I_5$
$\Delta(I_6,() = I_4$
$\Delta(I_7,F) = \{T \to T*F.\} = I_{10}$ (terminal pour règle 3)
$\Delta(I_7,nb) = I_5$
$\Delta(I_7,() = I_4$
$\Delta(I_8,)) = \{F \to (E).\} = I_{11}$ (terminal pour règle 5)

$\Delta(I_8, +) = \{E \to E + .T, T \to .T * F, T \to .F, F \to .nb, F \to .(E)\} = I_6$
$\Delta(I_9, *) = I_7$
OUFFFFFF ...!

Construction de la table d'analyse SLR :

1- Construire la collection d'items $\{I_o, \ldots, I_n\}$
2- l'état i est contruit à partir de I_i :
a) pour chaque $\Delta(I_i, a) = I_j$: mettre `decaller j` dans la case $M[i, a]$
b) pour chaque $\Delta(I_i, A) = I_j$: mettre `aller en j` dans la case $M[i, A]$
c) pour chaque $A \to \alpha.$ (sauf $A = S'$) contenu dans I_i :
mettre `reduire` $A \to \alpha$ dans **chaque** case $M[i, a]$ où $a \in$SUIVANT(A)
d) si $S' \to S. \in I_i$: mettre `accepter` dans la case $M[i, \$]$

Toujours le même exemple : il nous faut les SUIVANT (et PREMIER) :

	PREMIER	SUIVANT
E	nb (	$ +)
T	nb (	$ + *)
F	nb (	$ + *)

et donc la table d'analyse LR de cette grammaire est

état	nb	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				ACC			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Analyseur syntaxique : On part de l'état 0 et on empile et dépile non seulement les symboles (comme lors de l'analyseur LL) mais aussi les **états successifs**.

Exemple : l'analyse du mot $m = 3 + *4\$$ est donnée figure 5.4. La figure 5.5 donne l'analyse du mot $3 + 4 * 2\$$. On peut aussi dessiner l'arbre obtenu (figure 5.6)

pile	entrée	action
$\$ \boxed{0}$	$3 + *4\$$	d5
$\$ \boxed{0} 3 \boxed{5}$	$+ * 4\$$	r6 : $F \to$ nb
$\$ \boxed{0} F$	$+ * 4\$$	je suis en 0 avec F : je vais en 3
$\$ \boxed{0} F \boxed{3}$	$+ * 4\$$	r4 : $T \to F$
$\$ \boxed{0} T$	$+ * 4\$$	je suis en 0 avec T : je vais en 2
$\$ \boxed{0} T \boxed{2}$	$+ * 4\$$	r2 : $E \to T$
$\$ \boxed{0} E$	$+ * 4\$$	je suis en 0 avec E : je vais en 1
$\$ \boxed{0} E \boxed{1}$	$+ * 4\$$	d6
$\$ \boxed{0} E \boxed{1} + \boxed{6}$	$*4\$$	ERREUR !! Ce mot n'appartient pas au langage.

FIG. 5.4 - *Analyse LR du mot* $m = 3 + *4\$$

Remarques
• Cette méthode permet d'analyser plus de grammaires que la méthode descendante (car il y a plus de grammaires qui sont LR que de grammaires qui sont LL(1))
• En TP on va utiliser un outil (bison) qui construit tout seul une table d'analyse LR (LALR en fait, mais c'est presque pareil) à partir d'une grammaire donnée.
• Dans cette méthode d'analyse (LR), ça n'a strictement aucune importance que la grammaire soit récursive à gauche, même au contraire, elle **préfère**.

pile	entrée	action
\$ 0	$3+4*2\$$	d5
\$ 0 3 5	$+4*2\$$	r6 : $F \to$ nb
\$ 0 F	$+4*2\$$	je suis en 0 avec F : je vais en 3
\$ 0 F 3	$+4*2\$$	r4 : $T \to F$
\$ 0 T	$+4*2\$$	je suis en 0 avec T : je vais en 2
\$ 0 T 2	$+4*2\$$	r2 : $E \to T$
\$ 0 E	$+4*2\$$	je suis en 0 avec E : je vais en 1
\$ 0 E 1	$+4*2\$$	d6
\$ 0 E 1 + 6	$4*2\$$	d5
\$ 0 E 1 + 6 4 5	$*2\$$	r6 : $F \to$ nb
\$ 0 E 1 + 6 F	$*2\$$	je suis en 6 avec F : je vais en 3
\$ 0 E 1 + 6 F 3	$*2\$$	r4 : $T \to F$
\$ 0 E 1 + 6 T	$*2\$$	en 6 avec T : je vais en 9
\$ 0 E 1 + 6 T 9	$*2\$$	d7
\$ 0 E 1 + 6 T 9 * 7	$2\$$	d5
\$ 0 E 1 + 6 T 9 * 7 2 5	\$	r6 : $F \to$ nb
\$ 0 E 1 + 6 T 9 * 7 F	\$	en 7 avec F : je vais en 10
\$ 0 E 1 + 6 T 9 * 7 F 10	\$	r3 : $T \to T * F$
\$ 0 E 1 + 6 T	\$	en 6 avec T : je vais en 9
\$ 0 E 1 + 6 T 9	\$	r1 : $E \to E + T$
\$ 0 E	\$	en 0 avec E : je vais en 1
\$ 0 E 1	\$	ACCEPTÉ !!!

FIG. 5.5 - *Analyse LR du mot* $m = 3 + 4 * 2\$$

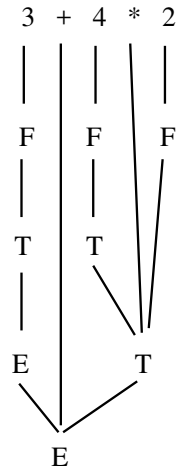


FIG. 5.6 - *arbre de dérivation du mot* $3 + 4 * 2$

• Les grammaires ambiguës provoquent des **conflits**
- conflit décallage/réduction : on ne peut pas décider à la lecture du terminal a s'il faut réduire une production $S \to \alpha$ ou décaller le terminal
- conflit réduction/réduction : on ne peut pas décider à la lecture du terminal a s'il faut réduire une production $S \to \alpha$ ou une production $T \to \beta$

On doit alors résoudre les conflits en donnant des **priorités** aux actions (décaller ou réduire) et aux productions.
Par exemple, soit la grammaire $E \to E + E | E * E |$nb
Soit à analyser $3 + 4 + 5$. Lorsqu'on lit le 2ième + on a le choix entre
- réduire ce qu'on a déja lu par $E \to E + E$. Ce qui nous donnera finalement le calcul $(3 + 4) + 5$
- décaller ce +, ce qui nous donnera finalement le calcul $3 + (4 + 5)$.

Ici on s'en cague car c'est pareil. Mais bon, + est associatif à gauche, donc on préfèrera réduire.
Soit à analyser $3 + 4 * 5$. Lorsqu'on lit le $*$ on a encore un choix shift/reduce. Si l'on réduit on calcule $(3 + 4) * 5$, si on décalle on calcule $3 + (4 * 5)$! On ne peut plus s'en foutre ! Il **faut** décaller.
Soit à analyser $3 * 4 + 5$. On ne s'en fout pas non plus, il faut réduire !
Bref, il faut mettre qlqpart dans l'analyseur le fait que $*$ est prioritaire sur +.
Tout ceci se voit dans la table d'analyse quand on essaye de la faire :

$$\begin{cases} (0) \quad S \to E & (2) \quad E \to E * E \\ (1) \quad E \to E + E & (3) \quad E \to \text{nb} \end{cases}$$

$I_0 = \{S \to .E, E \to .E + E, E \to .E * E, E \to .\text{nb}\}$
$\Delta(I_0, E) = \{S \to E., E \to E. + E, E \to E. * E\} = I_1$ **r0**
$\Delta(I_0, \text{nb}) = \{E \to \text{nb}.\} = I_2$ **r3**
$\Delta(I_1, +) = \{E \to E + .E, E \to .E + E, E \to .E * E, E \to .\text{nb}\} = I_3$
$\Delta(I_1, *) = \{E \to E * .E, E \to .E + E, E \to .E * E, E \to .\text{nb}\} = I_4$
$\Delta(I_3, E) = \{E \to E + E., E \to E. + E, E \to E. * E, \} = I_5$
$\Delta(I_3, \text{nb}) = I_2$
$\Delta(I_4, E) = \{E \to E * E., E \to E. + E, E \to E. * E, \} = I_6$
$\Delta(I_4, \text{nb}) = I_2$
$\Delta(I_5, +) = \Delta(I_6, +) = I_3$ et $\Delta(I_5, *) = \Delta(I_6, *) = I_4$
Comme Suivant(E)= $\{+, *, \$\}$ on obtient la table avec les conflits :

état	nb	+	*	$	E
0	d2				1
1		d3	d4	ACC	
2		r3	r3	r3	
3	d2				5
4	d2				6
5		d3 r1	d4 r1	r1	
6		d3 r2	d4 r2	r2	

C'est exactement ce qu'on vient de dire.

5.5 Erreurs syntaxiques

Beaucoup d'erreurs sont par nature syntaxiques (ou révélées lorsque les unités lexicales provenant de l'analyseur lexical contredisent les règles grammaticales). Le gestionnaire d'erreur doit
- indiquer la présence de l'erreur de façon claire et précise
- traiter l'erreur rapidement pour continuer l'analyse
- traiter l'erreur le plus efficacement possible de manière à ne pas en créer de nouvelles.

Heureusement, les erreurs communes (confusion entre deux séparateurs (par exemple entre ; et ,), oubli de ;, ...) sont simples et un mécanisme simple de traitement suffit en général. Cependant, une erreur peut se produire longtemps avant d'être détectée (par exemple l'oubli d'un { ou } dans un programme C). La nature de l'erreur est alors très difficile à déduire. La plupart du temps, le gestionnaire d'erreurs doit **deviner** ce que le programmeur avait en tête.
Lorsque le nombre d'erreur devient trop important, il est plus raisonnable de stopper l'analyse [4].

Il existe plusieurs stratégies de récupération sur erreur : mode panique, au niveau du syntagme [5], productions d'erreur, correction globale. Une récupération inadéquate peut provoquer une avalanche néfastes d'erreurs *illégitimes*, c'est à dire d'erreurs qui n'ont pas été faites par le programmeur mais sont la conséquence du changement

4. par exemple quel doit être le comportement d'un compilateur C confronté à un programme en Caml?
5. syntagme, priez pour nous ...

d'état de l'analyseur lors de la récupération sur erreur. Ces erreurs illégitimes peuvent être syntaxiques mais également sémantiques. Par exemple, pour se récupérer d'une erreur, l'analyseur syntaxique peut sauter la déclaration d'une variable. Lors de l'utilisation de cette variable, l'analyseur sémantique indiquera qu'elle n'a pas été déclarée.

5.5.1 Récupération en mode panique

C'est la méthode la plus simple à implanter. Quand il découvre une erreur, l'analyseur syntaxique élimine les symboles d'entrée les uns après les autres jusqu'à en rencontrer un qui appartienne à un ensemble d'unités lexicales de synchronisation, c'est à dire (par exemple) les délimiteurs (`;`, `end` ou `}`), dont le rôle dans un programme source est clair.
Bien que cette méthode saute en général une partie considérable du texte source sans en vérifier la validité, elle a l'avantage de la simplicité et ne peut pas entrer dans une boucle infinie.

5.5.2 Récupération au niveau du syntagme

Quand une erreur est découverte, l'analyseur syntaxique peut effectuer des corrections locales. Par exemple, remplacer une `,` par un `;`, un `wihle` par un `while`, insérer un `;` ou une `(`, ...Le choix de la modification à faire n'est pas évident du tout du tout en général. En outre, il faut faire attention à ne pas faire de modifications qui entrainerait une boucle infinie (par exemple décider d'insérer systèmatiquement un symbole juste avant le symbole courant).
L'inconvénient majeure de cette méthode est qu'il est pratiquement impossible de gérer les situations dans lesquelles l'erreur réelle s'est produite bien avant le point de détection.

On implante cette récupération sur erreur en remplissant les cases vides des tables d'analyse par des pointeurs vers des routines d'erreur. Ces routines remplacent, insérent ou suppriment des symboles d'entrée et émettent les messages appropriés.
Exemple : grammaire des expressions arithmétiques

$$\begin{cases} (1) \quad E \to E + E & (3) \quad E \to (E) \\ (2) \quad E \to E * E & (4) \quad E \to \text{nb} \end{cases}$$

La table d'analyse LR avec routines d'erreur est

état	nb	+	*	(	)	$	E
0	d3	**e1**	**e1**	d2	**e2**	**e1**	1
1	**e3**	d4	d5	**e3**	**e2**	ACC	
2	d3	**e1**	**e1**	d2	**e2**	**e1**	6
3	r4	r4	r4	r4	r4	r4	
4	d3	**e1**	**e1**	d2	**e2**	**e1**	7
5	d3	**e1**	**e1**	d2	**e2**	**e1**	8
6	**e3**	d4	d5	**e3**	d9	**e4**	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Les routines d'erreur étant :

e1 : (routine appelée depuis les états 0, 2, 4 et 5 lorsque l'on rencontre un opérateur ou la fin de chaîne d'entrée alors qu'on attend un opérande ou une parenthèse ouvrante)
Emettre le diagnostic `operande manquant`
Empiler un nombre quelconque et aller dans l'état 3

e2 : (routine appelée depuis les états 0,1,2,4 et 5 à la vue d'une parenthèse fermante)
Emettre le diagnostic `parenthese fermante excedentaire`
Ignorer cette parenthèse fermante

e3 : (routine appelée depuis les états 1 ou 6 lorsque l'on rencontre un nombre ou une parenthèse fermante alors que l'on attend un opérateur)
Emettre le diagnostic `operateur manquant`
Empiler `+` (par exemple) et aller à l'état 4

e4 : (routine appelée depuis l'état 6 qui attend un opérateur ou une parenthès fermante lorsque l'on rencontre la fin de chaîne)
Emettre le diagnostic `parenthese fermante oubliee`
Empiler une parenthèse fermante et aller à l'état 9

5.5.3 Productions d'erreur

Si l'on a une idée assez précise des erreurs courantes qui peuvent être rencontrées, il est possible d'augmenter la grammaire du langage avec des productions qui engendrent les constructions erronées.
Par exemple (pour un compilateur C) :

$I \rightarrow$ `if` $E\ I$	(erreur : il manque les parenthèses)
$I \rightarrow$ `if (` E `)` `then` I	(erreur : il n'y a pas de `then` en C)

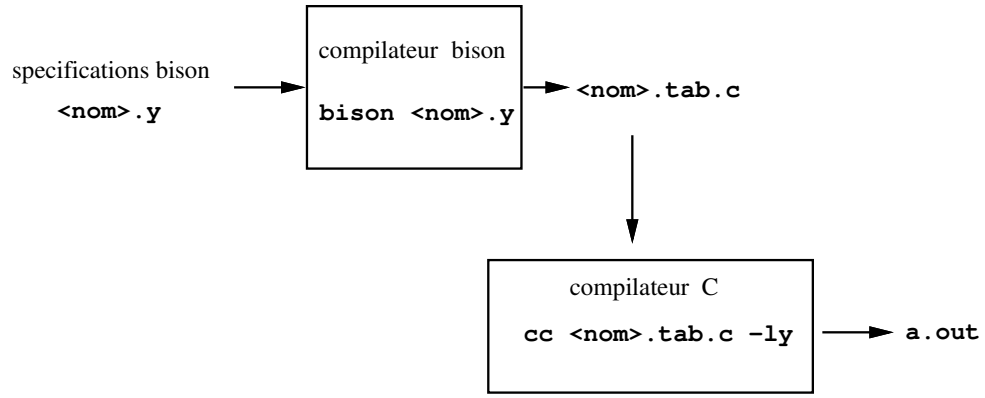
5.5.4 Correction globale

Dans l'idéal, il est souhaitable que le compilateur effectue aussi peu de changements que possible. Il existe des algorithmes qui permettent de choisir une séquence minimale de changements correspondant globalement au coût de correction le plus faible. Malheureusement, ces méthodes sont trop couteuses en temps et en espace pour être implantées en pratique et ont donc uniquement un interêt théorique. En outre, le programme correct le plus proche n'est pas forcemment celui que le programmeur avait en tête ...

Chapitre 6

L'outil `yacc/bison`

De nombreux outils ont été batis pour construire des analyseurs syntaxiques à partir de grammaires. C'est à dire des outils qui construisent automatiquement une table d'analyse à partir d'une grammaire donnée.
`yacc` est un utilitaire d'unix, `bison` est un produit gnu. `yacc/bison` accepte en entrée la description d'un langage sous forme de règles de productions et produit un **programme** écrit dans un langage de haut niveau (ici, le langage C) qui, une fois compilé, reconnait des phrases de ce langage (ce programme est donc un analyseur syntaxique).
`yacc` est l'acronyme de *Yet Another Compiler Compiler*, c'est à dire *encore un compilateur de compilateur*. Cela n'est pas tout à fait exact, `yacc/bison` tout seul ne permet pas d'écrire un compilateur, il faut rajouter une analyse lexicale (à l'aide de `(f)lex` par exemple) ainsi que des **actions sémantiques** pour l'analyse sémantique et la génération de code.



`yacc/bison` construit une table d'analyse LALR qui permet de faire une analyse ascendante. Il utilise donc le modèle **décallages-réductions**.

6.1 Structure du fichier de spécifications `bison`

```
%{
déclaration (en C) de variables, constantes, inclusions de fichiers, ...
%}
déclarations des unités lexicales utilisées
déclarations de priorités et de types
%%
règles de production et actions sémantiques
%%
routines C et bloc principal
```

• Les symboles terminaux utilisables dans la description du langage sont
  – des **unités lexicales** que l'on doit impérativement déclarer par `%token` *nom*. Par exemple :
    ```
    %token MC_sinon
    %token NOMBRE
    ```
  – des **caractères** entre quotes. Par exemple : `'+' 'a'`
• Les symboles non-terminaux sont les caractères ou les chaînes de caractères non déclarées comme unités lexicales.
`yacc/bison` fait la différence entre majuscules et minuscules. `SI` et `si` ne désignent pas le même objet.
• Les règles de production sont des suites d'instructions de la forme

```
non-terminal :  prod1
             |  prod2
                ...
             |  prodn
             ;
```

• Les **actions sémantiques** sont des instructions en C insérées dans les règles de production. Elles sont exécutées chaque fois qu'il y a réduction par la production associée.
Exemples :

```
G : S B 'X' {printf("mot reconnu");}
  ;
S : A {print("reduction par A");} T {printf("reduction par T");} 'a'
  ;
```

• La section du bloc principal doit contenir une fonction `yylex()` effectuant l'analyse lexicale du texte, car l'analyseur syntaxique l'appelle chaque fois qu'il a besoin du terminal suivant. On peut
- soit écrire cette fonction
- soit utiliser la fonction produite par un compilateur `(f)lex` appliqué à un fichier de spécifications `(f)lex` *nom*`.l`. Dans ce cas, il faut :
  - utiliser le compilateur `bison` du *nom*`.y` avec l'option `-d` qui produit un fichier *nom*`.tab.h` contenant les définitions (sous forme de `#define`) des unités lexicales (les `token`) rencontrées et du type `YYSTYPE` s'il est redéfini,
  - inclure le *nom*`.tab.h` au début du fichier de spécifications `(f)lex`
  - créer un `.o` à partir du `lex.yy.c` produit par `(f)lex`
  - lors de la compilation C du fichier *nom*`.tab.c`, il faut faire le lien avec ce `.o` et rajouter la bibliothèque (f)lex lors de la compilation C du fichier *nom*`.tab.c` avec : `gcc` *nom*`.tab.c lex.yy.o -ly -lfl` (attention à **l'ordre** de ces bibliothèques).

6.2 Attributs

A chaque symbole (terminal ou non) est associé une valeur (de type entier par défaut). Cette valeur peut être utilisée dans les actions sémantiques (comme un attribut **synthétisé**).
Le symbole `$$` référence la valeur de l'attribut associé au non-terminal de la partie gauche, tandis que `$i` référence la valeur associée au i-ème symbole (terminal ou non-terminal) ou **action sémantique** de la partie droite.
Exemple :

```
expr: expr '+' expr { tmp=$1+$3;} '+' expr { $$=tmp+$6;};
```

Par défaut, lorsqu'aucune action n'est indiquée, `yacc/bison` génère l'action {`$$=$1;`}

6.3 Communication avec l'analyseur lexical : `yylval`

L'analyseur syntaxique et l'analyseur lexical peuvent communiquer entre eux par l'intermédiaire de la variable `int yylval`.
Dans une action **lexicale** (donc dans le fichier `(f)lex` par exemple), l'instruction `return(`*unité*`)` permet de renvoyer à l'analyseur syntaxique l'unité lexicale *unité*. La **valeur** de cette unité lexicale peut être rangée dans `yylval`.
L'analyseur **syntaxique** prendra automatiquement le contenu de `yylval` comme valeur de l'attribut associé à cette unité lexicale.

La variable `yylval` est de type `YYSTYPE` (déclaré dans la bibliothèque `yacc/bison`) qui est un `int` par défaut. On peut changer ce type par un

```
#define YYSTYPE autre_type_C
```

ou encore par

```
%union { champs d'une union C }
```

qui déclarera automatiquement `YYSTYPE` comme étant une telle `union`.
Par exemple

```
%union {
          int entier;
          double reel;
          char * chaine;
       }
```

permet de stocker dans `yylval` à la fois des `int`, des `double` et des `char *`.
L'analyseur lexical pourra par exemple contenir

```
{nombre}  {
           yylval.entier=atoi(yytext);
           return NOMBRE;
          }
```

Le type des lexèmes doit alors être précisé en utilisant les noms des champs de l'union

```
%token <entier> NOMBRE
%token <chaine> IDENT CHAINE COMMENT
```

On peut également typer des non-terminaux (pour pouvoir associer une valeur de type autre que `int` à un non-terminal) par

```
%type <entier> S
%type <chaine> expr
```

6.4 Variables, fonctions et actions prédéfinies

`yyparse()` : appel de l'analyseur syntaxique.
`YYACCEPT` : instruction qui permet de stopper l'analyseur syntaxique.
`YYABORT` : instruction qui permet également de stopper l'analyseur, mais `yyparse()` retourne alors 1, ce qui peut être utilisé pour signifier l'echec de l'analyseur.
`main()` : le main par défaut se contente d'appeler `yyparse()`. L'utilisateur peut écrire son propre `main` dans la partie du bloc principal.
`%start` *non-terminal* : action pour signifier quel non-terminal est l'axiome. Par défaut, c'est le premier décrit dans les règles de production.

6.5 Conflits shift-reduce et reduce-reduce

Lorsque l'analyseur est confronté à des conflits, il rend compte du type et du nombre de conflits rencontrés :

```
>bison exemple.y
conflicts: 6 shift/reduce, 2 reduce/reduce
```

Il y a un conflit reduce/reduce lorsque le compilateur a le choix entre (au moins) deux productions pour **réduire** une chaîne. Les conflits shift/reduce apparaissent lorsque le compilateur a le choix entre **réduire** par une production et **décaller** le pointeur sur la chaîne d'entrée.
`yacc/bison` résoud les conflits de la manière suivante :

- conflit **reduce/reduce** : la production choisie est celle apparaissant en premier dans la spécification.
- conflit **shift/reduce** : c'est le shift qui est effectué.

Pour voir comment `bison` a résolu les conflits, il est nécessaire de consulter la table d'analyse qu'il a construit. Pour celà, il faut compiler avec l'option `-v`. Le fichier contenant la table s'appelle *nom*`.output`

6.5.1 Associativité et priorité des symboles terminaux

On peut essayer de résoudre soit même les conflits (ou tout du moins préciser comment on veut les résoudre) en donnant des associativités (droite ou gauche) et des priorités aux symboles terminaux.
Les déclarations suivantes (dans la section des définitions)

```
%left term1 term2
%right term3
%left term4
%nonassoc term5
```

indiquent que les symboles terminaux *term1*, *term2* et *term4* sont associatifs à gauche, *term3* est associatif à droite, alors que *term5* n'est pas associatif.
Les priorités des symboles sont données par l'ordre dans lequel apparait leur déclaration d'associativité, les premiers ayant la plus faible priorité. Lorsque les symboles sont dans la même déclaration d'associativité, ils ont la même priorité.
La priorité (ainsi que l'associativité) d'une production est définie comme étant celle de son terminal le plus à droite. On peut forcer la priorité d'une production en faisant suivre la production de la déclaration

```
%prec terminal-ou-unite-lexicale
```

ce qui a pour effet de donner comme priorité (et comme associativité) à la production celle du *terminal-ou-unite-lexicale* (ce *terminal-ou-unite-lexicale* devant être défini, même de manière factice, dans la partie Ib).
Un conflit **shift/reduce**, i.e. un choix entre entre une réduction $A \to \alpha$ et un décallage d'un symbole d'entrée a, est alors résolu en appliquant les règles suivantes :

- si la priorité de la production $A \to \alpha$ est supérieure à celle de a, c'est la réduction qui est effectuée

- si les priorités sont les mêmes et si la production est associative à gauche, c'est la réduction qui est effectuée
- dans les autres cas, c'est le shift qui est effectué.

6.6 Récupération des erreurs

Lorsque l'analyseur produit par bison rencontre une erreur, il appelle par défaut la fonction `yyerror(char *)` qui se contente d'afficher le message `parse error`, puis il s'arrête. Cette fonction peut être redéfinie par l'utilisateur. Il est possible de traiter de manière plus explicite les erreurs en utilisant le mot clé bison `error`.
On peut rajouter dans toute production de la forme $A \rightarrow \alpha_1|\alpha_2|\ldots$ une production

$A \rightarrow$ `error` β

Dans ce cas, une production d'erreur sera traitée comme une production classique. On pourra donc lui associer une action sémantique contenant un message d'erreur.
Dès qu'une erreur est rencontrée, tous les caractères sont avalés jusqu'à rencontrer le caractère correspondant à β.
Exemple : La production

instr→ `error` ;

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

6.7 Exemples de fichier .y

Cet exemple reconnait les mots qui ont un nombre pair de a et impair de b. Le mot doit se terminer par le symbole $. Cet exemple ne fait pas appel à la fonction `yylex` générée par le compilateur `(f)lex`.

```
%%
mot : PI '$'  {printf("mot accepte\n");YYACCEPT;}
       ;

PP : 'a' IP
   | 'b' PI
   | /* vide */
   ;
IP : 'a' PP
   | 'b' II
   | 'a'
   ;
PI : 'a' II
   | 'b' PP
   | 'b'
   ;
II : 'a' PI
   | 'b' IP
   | 'a' 'b'
   | 'b' 'a'
   ;
%%
int yylex() {
  char car=getchar();
  if (car=='a' || car=='b' || car=='$')  return(car);
  else printf("ERREUR : caractere non reconnu : %c ",car);
}
```

Ce deuxième exemple lit des listes d'entiers précédées soit du mot `somme`, soit du mot `produit`. Une liste d'entiers est composée d'entiers séparés par des virgules et se termine par un point. Lorsque la liste débute par `somme`, l'exécutable doit afficher la somme des entiers, lorsqu'elle débute par `produit`, il doit en afficher le produit. Le fichier doit se terminer par $.
Cet exemple utilise la fonction `yylex` générée par le compilateur `flex` à partir du fichier de spécification `exemple2.l` suivant

```
%{
#include <stdlib.h>
#include "exemple2.tab.h"      // INCLUSION DES DEFS DES TOKEN
int  nbligne=0;
%}
chiffre    [0-9]
entier     {chiffre}+
espace     [ \t]
%%
somme       return(SOMME);
produit     return(PRODUIT);
\n          nbligne++;
[.,]        return(yytext[0]);
{entier}    {
              yylval=atoi(yytext);
              return(NOMBRE);
            }
{espace}+   /* rien */;
"$"         return(FIN);
.           printf("ERREUR ligne %d : %c inconnu\n",nbligne,yytext[0]);
%%
```

Le fichier de spécifications `bison` est alors le suivant

```
%token SOMME
%token PRODUIT
%token NOMBRE
%token FIN
%%
texte : liste texte
      | FIN {printf("Merci et a bientot\n");YYACCEPT;}
      ;

liste : SOMME sentiers '.' {printf("la somme est %d\n",$2);}
      | PRODUIT pentiers '.' {printf("le produit est %d\n",$2);}
      ;

sentiers : sentiers ',' NOMBRE {$$=$1+yylval;}
         | NOMBRE {$$=yylval;}
         ;

pentiers : pentiers ',' NOMBRE {$$=$1*yylval;}
         | NOMBRE {$$=yylval;}
         ;

%%
```

Un Makefile pour compiler tout ça est

```
exemple2 : lex.yy.o exemple2.tab.c
	gcc exemple2.tab.c lex.yy.o -o exemple2 -ly -lfl
lex.yy.o :   lex.yy.c
	gcc -c lex.yy.c
lex.yy.c : exemple2.l exemple2.tab.h
	flex exemple2.l
exemple2.tab.h :  exemple2.y
	bison -d  exemple2.y
exemple2.tab.c : exemple2.y
	bison -d exemple2.y
```

Chapitre 7

Théorie des langages : les automates

7.1 Classification des grammaires

Nous avons déja vu qu'il y avait des langages *réguliers* et d'autres non réguliers. On distingue en fait quatres types de langage, correspondant à quatres types de grammaires. Ces différents types dépendent de la forme des productions.

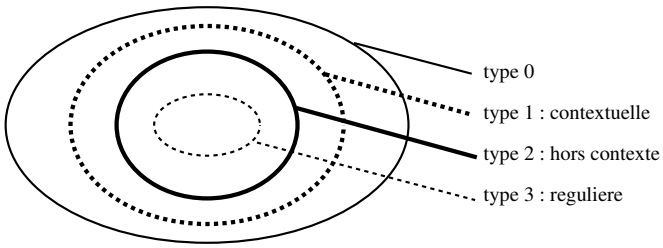
• Grammaire de **type 3 (régulière)**
Les productions sont de la forme $A \to wB$ ou $A \to w$ avec $A \in V_N$ (un seul symbole non terminal), $B \in V_N$ et $w \in V_T^*$ (une chaîne de symboles terminaux)
Exemple : $\begin{cases} S \to aS|T|abU|cc \\ T \to cT|\varepsilon \\ U \to abS \end{cases}$

• Grammaire de **type 2 (hors contexte) (algébrique)**
Les productions sont de la forme $A \to \beta$ avec $A \in V_N$ et $\beta \in (V_N \bigcup V_T)^*$
Exemple (grammaire de Dyck) : $S \to (\ S\ )\ S|\varepsilon$

• Grammaire de **type 1 (contextuelle)**
Les productions sont de la forme $\alpha \to \beta$ avec $\alpha \in (V_N \bigcup V_T)^+$ et $\beta \in (V_N \bigcup V_T)^*$ et $|\alpha| \leq |\beta|$
Les productions $S \to \varepsilon$ sont autorisées si $S \in V_N$ n'apparait pas dans la partie droite d'une production de G.
Exemple : $\begin{cases} \cdots \\ a\ A\ b\ \to a\ truc\ b \end{cases}$
Le remplacement de A par *truc* ne se fait que dans le **contexte** précis où A est entouré par les symboles a et b

• Grammaire de **type 0** : aucune restrictions sur les règles
Les productions sont de la forme $\alpha \to \beta$ avec $\alpha \in (V_N \bigcup V_T)^+$ et $\beta \in (V_N \bigcup V_T)^*$

Propriété 7.1 *Les grammaires de type 0 englobent les grammaires de type 1 qui englobent les grammaires de type 2 qui englobent les grammaires de type 3.*



7.1.1 Les langages contextuels

Exemples de langages contextuels (mais non hors contexte) [1] :

• Exemple 1 : L=$\{wdw, w \in \{a,b,c\}^*\}$ n'est pas hors contexte, c'est à dire qu'il n'existe pas de grammaire hors contexte permettant de le décrire. Ce langage est une traduction abstraite du problème consistant à controler

1. par la suite, on dira qu'un langage est de type i s'il est de type i et pas de type $i+1$

que, dans un programme, une variable doit être déclarée avant d'être utilisée.

Une grammaire (contextuelle) générant ce langage?
Commencons par générer $wd\tilde{w}$, on sait faire c'est hors contexte ça :

$$\{\ S \to aSa|bSb|cSc|d$$

Mais ça ne résoud pas notre problème. Nous on veut ensuite déplacer chaque lettre du $\tilde{w}$. Pour obliger à déplacer, mettons plutôt des non-terminaux (les terminaux permettent de finir et donc autoriserait qu'on s'arrête avant d'avoir déplacé tout le monde), et ceci entre deux marqueurs de début et fin :

$$\begin{cases} S' \to SF \\ S \to aSA|bSB|cSC|dD \end{cases}$$

On obtient par exemple des mots du genre $abbcadDACBBAF$.
Donnons alors un coup de baguette magique sur chaque symbole qui doit être déplacé :

$$\begin{cases} DA \to DA' \\ DB \to DB' \\ DC \to DC' \end{cases}$$

puis déplaçons les :

$$\begin{cases} A'A \to AA' & A'B \to BA' & A'C \to CA' \\ B'A \to AB' & A'B \to BB' & B'C \to CB' \\ C'A \to AC' & A'B \to BC' & C'C \to CC' \end{cases}$$

Sur notre mot, cela nous pousse à faire : $abbacadDACABBAF \to abbcadDA'CBBAF \xrightarrow{*} abbcadDCBBAA'F$ (le A est arrivé à sa bonne place), et les autres coups de baguettes magiques déplacent de même le C ($abbcadDCBBAA'F \to abbcadDC'BBAA'F \xrightarrow{*} abbcadDBBAC'A'F$) puis les autres lettres ($abbcadDBBAC'A'F \xrightarrow{*} abbcadDA'B'B'C'A'F$)
Quand ils sont arrivés à leur place, on les remet normaux (terminaux) :

$$\begin{cases} A'F \to Fa \\ B'F \to Fb \\ C'F \to Fc \end{cases}$$

Sur le mot en exemple, ça nous permet donc d'avoir :
$abbcadDA'B'B'C'A'F \to abbcadDA'B'B'C'Fa \to abbcadDA'B'B'Fca \xrightarrow{*} abbcadDFabbca$
Il ne reste plus qu'à virer les F et D quand tout est fini. Conclusion : superbe grammaire contextuelle

$$\begin{cases} S' \to SF & & \\ S \to aSA|bSB|cSC|dD & & \\ DA \to DA' & DB \to DB' & DC \to DC' \\ A'A \to AA' & A'B \to BA' & A'C \to CA' \\ B'A \to AB' & A'B \to BB' & B'C \to CB' \\ C'A \to AC' & A'B \to BC' & C'C \to CC' \\ A'F \to Fa & B'F \to Fb & C'F \to Fc \\ DF \to \varepsilon & & \end{cases}$$

On a une grammaire contextuelle, donc le langage est contextuel[2].

• Exemple 2 : L=$\{a^n b^m c^n d^m, n \geq 1, m \geq 1\}$ est contextuel. Ce langage abstrait le problème de la vérification de la correspondance entre paramètres formels et effectifs d'une fonction.

7.1.2 Les langages réguliers

Les langages réguliers peuvent être générés par une grammaire régulière. Donc toute ER peut s'exprimer par une grammaire, et même par une grammaire régulière.

Exemples :
• $S \to aS|bS|\varepsilon$ génère l'expression $(a|b)^*$
• La grammaire $\begin{cases} S \to a|Bc \\ B \to Bb|\varepsilon \end{cases}$ génère $a|b^*c$ mais n'est pas régulière. Une grammaire régulière équivalente (qui

2. précisons le, car il aurait pu être carrement de type 0

génère le même langage) est $\begin{cases} S \to a|S' \\ S' \to bS'|c \end{cases}$

• L'ER $(a|b)^*abb(a|b)^*$ est générée par la grammaire non régulière $\begin{cases} S \to TabbT \\ T \to aT|bT|\varepsilon \end{cases}$ ou encore par la grammaire régulière $\begin{cases} S \to aS|bS|T \\ T \to abbU \\ U \to \varepsilon|aU|bU \end{cases}$

• $b^*ab^*ab^*ab^*$ est générée par la grammaire non régulière $\begin{cases} S \to BaBaBaB \\ B \to bB|\varepsilon \end{cases}$ équivalente à la grammaire régulière $\begin{cases} S \to bS|T & W \to bW|X \\ T \to aU & X \to aY \\ U \to bU|V & Y \to \varepsilon|bY \\ V \to aW \end{cases}$

• $\begin{cases} S \to TaaU \\ T \to abbcT|babaT|abbc|baba \\ U \to ccU|bbU|\varepsilon \end{cases}$ (qui est non régulière) génère $(abbc|baba)^+aa(cc|bb)*$

7.1.3 Les reconnaisseurs

Le problème qui se pose est de pouvoir reconnaître si un mot donné appartient à un langage donné. Un reconnaisseur pour un langage est un programme qui prend en entrée une chaîne x et répond oui si x est une phrase (un mot) du langage et non sinon. C'est un analyseur syntaxique quoi !

Théorème 7.2 *Les **automates à états finis** sont des reconnaisseurs pour les langages réguliers.*

Théorème 7.3 *Les **automates à pile** sont des reconnaisseurs pour les langages hors contexte.*

Les grammaires hors contexte et les grammaires régulières jouent donc un rôle particulièrement important en informatique (puisqu'il existe des analyseurs syntaxiques pour elles). Malheureusement, la plupart des langages de programmation usuels ne peuvent être complètement décrits par une grammaire hors contexte (et donc encore moins par une grammaire régulière). Les parties sensibles au contexte (règles de compatibilité de types, correspondance entre le nombre d'arguments formels et effectifs lors de l'appel d'une procédure, ...) sont généralement décrites de manière informelle à coté de la grammaire du langage qui, elle, est hors contexte. Ainsi, lors de la réalisation d'un compilateur, ces deux aspects sont aussi séparés : l'analyse syntaxique s'appuie sur la grammaire (hors contexte) et les contraintes contextuelles sont rajoutées par la suite.

7.2 Automates à états finis

Définition 7.4 *Un **automate à états finis (AEF)** est défini par*

- *un ensemble **fini** E d'états*
- *un état e_0 distingué comme étant l'**état initial***
- *un ensemble fini T d'états distingués comme **états finaux** (ou **états terminaux**)*
- *un alphabet Σ des symboles d'entrée*
- *une **fonction de transition** Δ qui à tout couple formé d'un état et d'un symbole de Σ fait correspondre un ensemble (éventuellement vide) d'états : $\Delta(e_i, a) = \{e_{i_1}, \ldots, e_{i_n}\}$*

Exemple : $\Sigma = \{a, b\}$, $E = \{0, 1, 2, 3\}$, $e_0 = 0$, $T = \{3\}$
$\Delta(0, a) = \{0, 1\}$, $\Delta(0, b) = \{0\}$, $\Delta(1, b) = \{2\}$, $\Delta(2, b) = \{3\}$ (et $\Delta(e, l) = \emptyset$ sinon)
Représentation graphique :

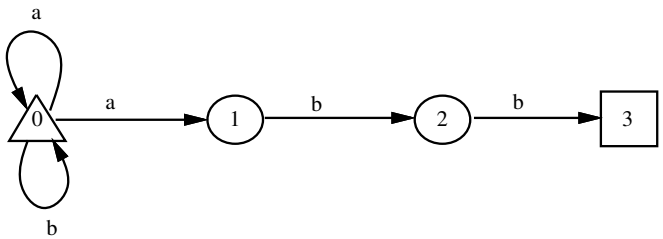


FIG. 7.1 - *Un exemple d'automate*

Convention : on dessinera un triangle pour l'état initial et un carré pour les états finaux.

Représentation par une **table de transition**:

état	a	b
0	0,1	0
1	-	2
2	-	3
3	-	-

$e_0 = 0$ et $T = \{3\}$

Attention à toujours bien préciser l'état initial et le ou les états finaux/terminaux.

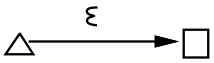
Définition 7.5 *Le **langage reconnu par un automate** est l'ensemble des chaînes qui permettent de passer de l'état initial à un état terminal.*

L'automate de la figure 7.1 accepte le langage (régulier) (l'expression régulière) $(a|b)^*abb$
Un automate peut très facilement être simulé par un algorithme (et donc on peut écrire un programme simulant un automate fini). C'est encore plus facile si l'automate est **déterministe**, c'est à dire lorsqu'il n'y a pas à choisir entre 2 transitions. Ce que signifie donc le théorème 7.2 c'est que l'on peut écrire un programme reconnaissant tout mot (toute phrase) de tout langage régulier. Ainsi, si l'on veut faire l'analyse lexicale d'un langage régulier, il suffit d'écrire un programme simulant l'automate qui lui est associé.

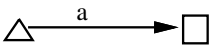
7.2.1 Construction d'un AEF à partir d'une E.R.

(Construction de Thompson, basée sur la récursivité des expressions régulières)
Pour une expression régulière s, on note A(s) un automate reconnaissant cette expression.

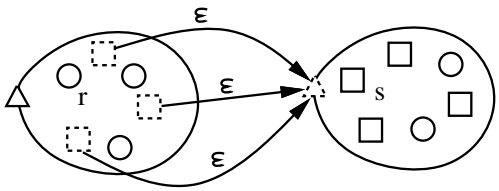
- automate acceptant la chaîne vide



- automate acceptant la lettre a

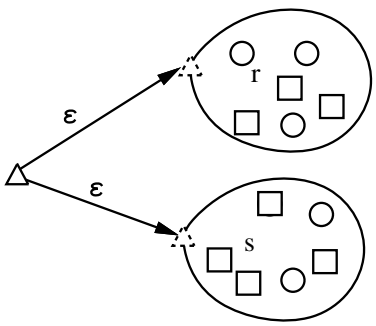


- automate acceptant $(r)(s)$



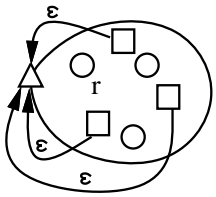
1. mettre une ε-transition de chaque état terminal de A(r) vers l'état initial de A(s)
2. les états terminaux de A(r) ne sont plus terminaux
3. le nouvel état initial est celui de A(r)
4. (l'ancien état initial de A(s) n'est plus état initial)

- automate reconnaissant $r|s$



1. créer un nouvel état initial q
2. mettre une ε-transition de q vers les états initiaux de A(r) et A(s)
3. (les états initiaux de A(r) et A(s) ne sont plus états initiaux)

– automate reconnaissant r^+



mettre des ε-transition de chaque état terminal de A(r) vers son état initial

Exemple : $(ab|bca)^+bb(ab|cb)^*a$ correspond à l'automate donné figure 7.2. Mais on peut bien sûr le simplifier et virer pas mal d'ε-transitions, pour obtenir par exemple celui donné figure 7.3.

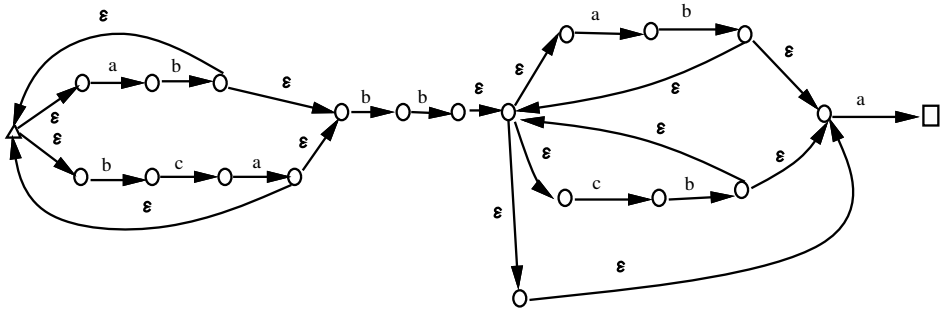


FIG. 7.2 - *Automate reconnaissant* $(ab|bca)^+bb(ab|cb)^*a$

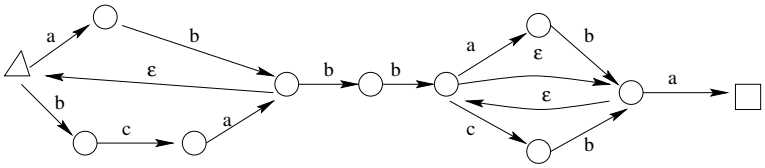


FIG. 7.3 - *Automate simplifié reconnaissant* $(ab|bca)^+bb(ab|cb)^*a$

7.2.2 Automates finis déterministes (AFD)

Définition 7.6 *On appelle* ε***-transition***, *une transition par le symbole* ε *entre deux états.*

Définition 7.7 *Un automate fini est dit* ***déterministe*** *lorsqu'il ne possède pas de* ε*-transition et lorsque pour chaque état* e *et pour chaque symbole* a*, il y a* ***au plus un*** *arc étiqueté* a *qui quitte* e

L'automate donné en exemple figure 7.1 n'est pas déterministe, puisque de l'état 0, avec la lettre a, on peut aller soit dans l'état 0 soit dans l'état 1.

Les AFD sont plus faciles à simuler (pas de choix dans les transitions, donc jamais de retours en arrière à faire). Il existe des algorithmes permettant de **déterminiser** un automate non déterministe (c'est à dire de construire un AFD qui reconnait le même langage que l'AFN[3] donné). L'AFD obtenu comporte en général plus d'états que l'AFN, donc le programme le simulant occupe plus de mémoire. Mais il est plus rapide.

Déterminisation d'un AFN qui ne contient pas d'ε-transitions

Principe : considérer des ensembles d'états plutôt que des états.

1. Partir de l'état initial
2. Rajouter dans la table de transition tout les nouveaux "états" produits, avec leurs transitions
3. Recommencer 2 jusqu'à ce qu'il n'y ait plus de nouvel "état"
4. Tous les "états" contenant au moins un état terminal deviennent terminaux
5. Renuméroter alors les états.

3. AFN pour *Automate Fini Non déterministe*

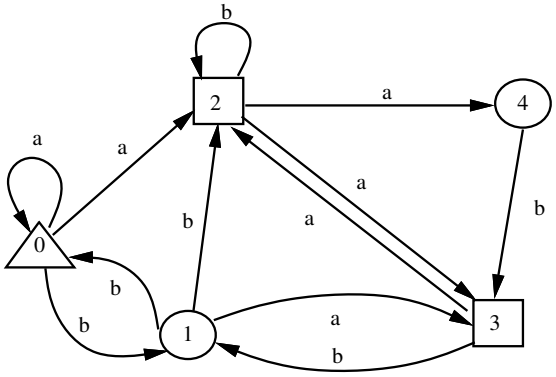


FIG. 7.4 - *Automate non déterministe d'un langage Alfred*

Exemple : appliquons l'algorithme sur l'automate représenté figure 7.4

"état"	a	b
0	0,2	1
0,2	0,2,3,4	1,2
1	3	0,2
0,2,3,4	0,2,3,4	1,2,3
1,2	3,4	0,2
3	2	1
1,2,3	2,3,4	0,1,2
3,4	2	1,3
2	3,4	2
2,3,4	2,3,4	1,2,3
0,1,2	0,2,3,4	0,1,2
1,3	2,3	0,1,2
2,3	2,3,4	1,2

=

état	a	b
0	1	2
1	3	4
2	5	1
3	3	6
4	7	1
5	8	2
6	9	10
7	8	11
8	7	8
9	9	6
10	3	10
11	12	10
12	9	4

$e_0 = 0$ et
$T = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
(voir figure 7.5)

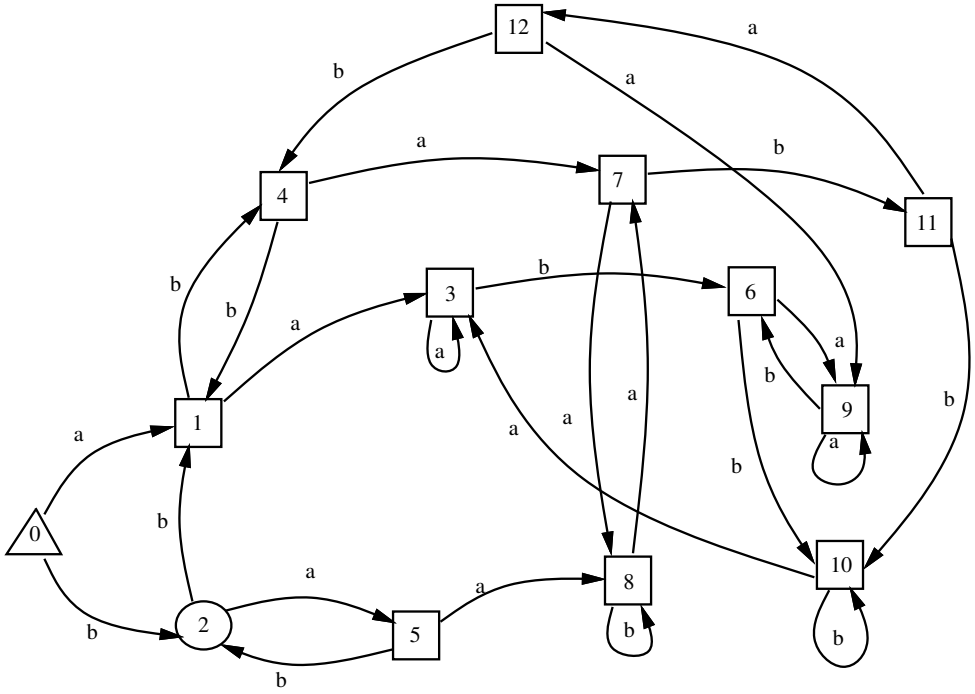


FIG. 7.5 - *Automate déterministe pour Alfred*

Déterminisation d'un AFN (cas général)

Principe : considérer l'ε-fermeture des ensembles d'états

Définition 7.8 *On appelle* ***ε-fermeture*** *de l'ensemble d'états $T = \{e_1, \ldots, e_n\}$ l'ensemble des états accessibles depuis un état e_i de T par des ε-transitions*

Calcul de l'ε-fermeture de $T = \{e_1, \ldots, e_n\}$:

```
Mettre tous les états de T dans une pile P
Initialiser ε-fermeture(T) à T
Tant que P est non vide faire
    Soit p l'état en sommet de P
    dépiler P
    Pour chaque état e tel qu'il y a une ε-transition entre p et e faire
        Si e n'est pas déja dans ε-fermeture(T)
            ajouter e à ε-fermeture(T)
            empiler e dans P
        finsi
    finpour
fin tantque
```

Exemple.

Soit l'AFN

état	a	b	c	ε
0	2	-	0	1
1	3	4	-	-
2	-	-	1,4	0
3	-	1	-	-
4	-	-	3	2

$e_0 = 0$ et $T = \{4\}$

On a par exemple ε-fermeture$(\{0\}) = \{0, 1\}$, ε-ferm$(\{1, 2\}) = \{1, 2, 0\}$, ε-ferm$(\{3, 4\}) = \{3, 4, 0, 1, 2\}, \ldots$

Déterminisation :

1. Partir de l'ε-**fermeture** de l'état initial
2. Rajouter dans la table de transition toutes les ϵ-**fermetures** des nouveaux "états" produits, avec leurs transitions
3. Recommencer 2 jusqu'à ce qu'il n'y ait plus de nouvel "état"
4. Tous les "états" contenant au moins un état terminal deviennent terminaux
5. Renuméroter alors les états.

Sur le dernier exemple, on obtient

état	a	b	c
0,1	2,3,0,1	4,2,0,1	0,1
0,1,2,3	0,1,2,3	0,1,2,4	0,1,4,2
0,1,2,4	0,1,2,3	0,1,2,4	0,1,3,4,2
0,1,2,3,4	0,1,2,3	0,1,2,4	0,1,2,3,4

=

état	a	b	c
0	1	2	0
1	1	2	2
2	1	2	3
3	1	2	3

avec $e_0 = 0$ et $T = \{2, 3\}$

Autre exemple : cherchons un AFD reconnaissant l'ER $(a^*b)^*a^*(c(ac)^*(aa)^*a)^+$.
On a l'AFN donné figure 7.6. Appliquons l'algorithme de déterminisation.

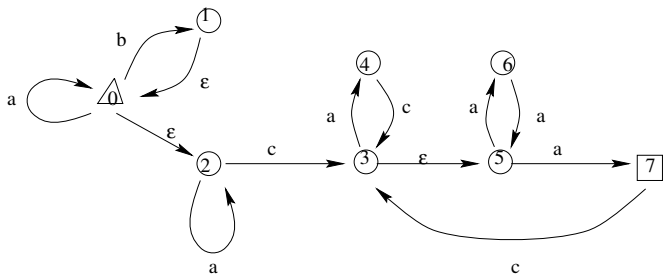


FIG. 7.6 - *Un AFN pour* $(a^*b)^*a^*(c(ac)^*(aa)^*a)^+$

Attention, il faut partir de l'ε-fermeture de l'état initial !

	a	b	c
0,2	0,2	1,0,2	3,5
0,1,2	0,2	0,1,2	3,5
3,5	4,6,7	-	-
4,6,7	5	-	3,5
5	6,7	-	-
6,7	5	-	3,5

=

	a	b	c
0	0	1	2
1	0	1	2
2	3	-	-
[3]	4	-	2
4	5	-	-
[5]	4	-	2

$e_0 = 0$
(cf figure 7.7)

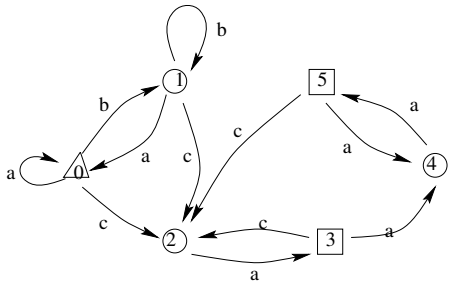


FIG. 7.7 - *Un AFD pour* $(a^*b)^*a^*(c(ac)^*(aa)^*a)^+$

7.2.3 Minimisation d'un AFD

But : obtenir un automate ayant le minimum d'états possible. En effet, certains états peuvent être équivalents. Par exemple on peut facilement remarquer que dans l'automate de la figure 7.7 (ou cf la table de transitions, ça se voit mieux) les états 2 et 4 sont équivalents, 0 et 1 sont équivalents, et enfin 3 et 5 sont équivalents. On a donc l'AFD minimal de la figure 7.8. C'est amusant car ça veut donc dire que $(a^*b)^*a^*(c(ac)^*(aa)^*a)^+ = (a|b)^*ca(ca|aa)^*$. Eh oui quand on y réflechit un peu ...

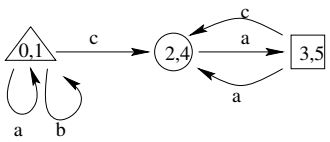


FIG. 7.8 - *l'AFD minimal pour* $(a^*b)^*a^*(c(ac)^*(aa)^*a)^+$

Principe de minimisation: on définit des classes d'équivalence d'états par raffinements successifs. Chaque classe d'équivalence obtenue forme un seul même état du nouvel automate.

1 - Faire deux classes : A contenant les états terminaux et B contenant les états non terminaux.
2 - S'il existe un symbole a et deux états e_1 et e_2 d'une même classe tels que $\Delta(e_1, a)$ et $\Delta(e_2, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer e_1 et e_2.
3 - Recommencer 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
4 - Chaque classe restante forme un état du nouvel automate

Minimisons l'AFD de la figure 7.5 :

(1) $A = \{0, 2\}$ et $B = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

(2) • $\Delta(0, a)$ et $\Delta(2, a)$ n'appartiennent pas à la même classe, alors A se casse en $A = \{0\}$ et $C = \{2\}$

• $\Delta(5, b)$ et $\Delta(1, b)$ n'appartiennent pas à la même classe, alors B se casse en $D = \{5\}$ et $B = \{1, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$

(3) Ça ne bouge plus.

On obtient donc l'automate à 4 états (correspondants aux 4 classes) de la figure 7.9.

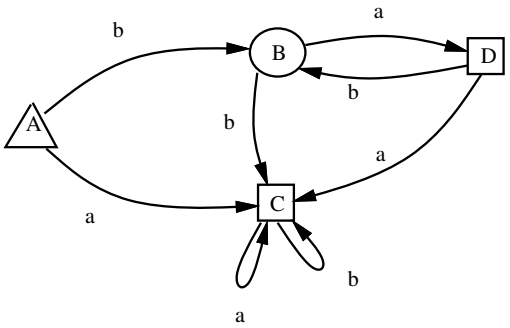


FIG. 7.9 - *Automate minimum pour le langage Alfred*

7.2.4 Calcul d'une E.R. décrite par un A.E.F.

On a dit qu'un AEF et une ER c'était pareil, on a vu comment passer d'une ER à une AEF, maintenant il reste à passer d'un AEF à une ER.

On appelle L_i le langage que reconnaîtrait l'automate si e_i était son état initial. On peut alors écrire un système d'équations liant tous les L_i :

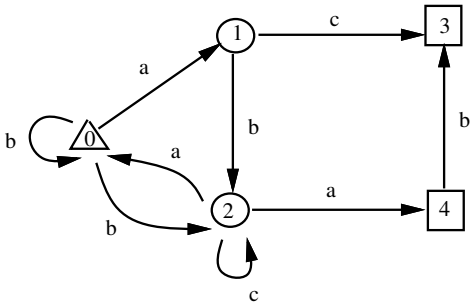
- toutes les transitions de l'état e_i ($\Delta(e_i, a_1) = e_{j_1}, \ldots, \Delta(e_i, a_p) = e_{j_p}$) permettent d'écrire l'équation $L_i = a_1 L_{j_1} | a_2 L_{j_2} | \ldots | a_p L_{j_p}$
- pour chaque $e_i \in T$ on rajoute $L_i = \ldots | \varepsilon$

On résoud ensuite le système (on cherche à calculer L_0) en remarquant juste que

Propriété 7.9
si $L = \alpha L | \beta$ *alors* $L = \alpha^* \beta$

• **Attention:** si on a $L = \alpha L$, ça n'implique **absolument pas** que $L = \alpha^*$. Ca indique juste qu'on est dans une impasse (on boucle) et donc on a dû se planter quelquepart.
• si on a $L = aL|bL|c$, ça s'écrit encore $L = (a|b)L|c$ ce qui implique que $L = (a|b)^*c$. On pourrait faire aussi : $L = aL|bL|c = aL|(bL|c) \Rightarrow L = a^*(bL|c) = (a^*bL)|(a^*c) \Rightarrow L = (a^*b)^*a^*c$ ce qui exprime le même langage mais en moins lisible [4]

Exemple complet : soit l'automate



on obtient le système

$$\begin{cases} L_0 = bL_0|aL_1|bL_2 \\ L_1 = bL_2|cL_3 \\ L_2 = aL_0|cL_2|aL_4 \\ L_3 = \varepsilon \\ L_4 = \varepsilon|bL_3 \end{cases}$$

$$\Leftrightarrow \begin{cases} L_3 = \varepsilon \\ L_4 = \varepsilon|b \\ L_2 = aL_0|cL_2|a(\varepsilon|b) = aL_0|cL_2|a|ab = c^*(a|ab|aL_0) \\ L_1 = bL_2|c = c|bc^*(a|ab|aL_0) \\ L_0 = bL_0|ac|abc^*(a|ab|aL_0)|bc^*(a|ab|aL_0) \end{cases}$$

$$\Rightarrow L_0 = ac|(a|\varepsilon)bc^*(a|ab)|(a|\varepsilon)bc^*aL_0|bL_0$$

$$\Leftrightarrow L_0 = (((a|\varepsilon)bc^*a)|b)^*(ac|(a|\varepsilon)bc^*(a|ab))$$

REMARQUE:
A une expression régulière peut correspondre plusieurs AEF. A un AEF peut correspondre plusieurs expressions regulières. Mais à chaque fois c'est en fait le(la) même AEF (expression) mais sous des écritures différentes. En particulier, c'est toujours le **même unique langage** qui est décrit.

7.3 Les automates à piles

Les langages hors contextes ne sont pas reconnaissables par un AEF. Par exemple, il n'existe pas d'AEF (ni d'ER) reconnaissant $\{a^n b^n, n \geq 0\}$. Pour les reconnaitre, on a besoin d'un outil plus puissant : les automates à pile.

Une transition d'un AEF est de la forme $\Delta(p, a) = q$, c'est à dire que si on est dans un état p et qu'on lit la lettre a, on passe dans l'état q.
Dans un automate à pile, on rajoute une pile [5]. Les transitions demandent donc en plus une condition sur l'état de la pile (sur la valeur qu'il y a en sommet de pile) et indiquent comment modifier la pile. Elles sont de la forme $\Delta(p, a, \alpha) = (q, \beta)$, c'est à dire que si on est dans l'état p avec le mot α en sommet de pile et la lettre a en entrée, on peut passer dans l'état q en remplaçant α par β dans la pile.

Définition 7.10 *Un* ***automate à pile*** *est défini par :*

4. certes, c'est subjectif ...
5. noooon?

- *un ensemble fini d'état* E
- *un état initial* $e_o \in E$
- *un ensemble fini d'états finaux* T
- *un alphabet* Σ *des symboles d'entrée*
- *un alphabet* Γ *des symboles de pile*
- *un symbole* $\tau \in \Gamma$ *de fond de pile*
- *une relation de transition* $\Delta : E \times \Sigma^* \times \Gamma^* \to E \times \Gamma^*$

Définition 7.11 *Le* ***langage reconnu par un automate à pile*** *est l'ensemble des chaînes qui permettent de passer de l'état initial à un état final en démarrant avec une pile contenant le symbole de fond de pile et en terminant avec une pile vide*

Exemple : automate à pile reconnaissant $\{a^n b^n, n \geq 0\}$

$E = \{s, p, q\}$ $\quad e_0 = s$ $\quad T = \{q\}$

$\Sigma = \{a, b\}$ $\quad \Gamma = \{A, \$\}$ $\quad \tau = \$$

Transitions :

$$\begin{cases} (s, a, \varepsilon) \to (s, A) \\ (s, b, A) \to (p, \varepsilon) \\ (p, b, A) \to (p, \varepsilon) \\ (s, \varepsilon, \$) \to (q, \varepsilon) \\ (p, \varepsilon, \$) \to (q, \varepsilon) \end{cases}$$

Soit le mot $aaabbb$ en entrée :

pile	entrée	état	transition effectuée
$\$$	$aaabbb$	s	$(s, a, \varepsilon) \to (s, A)$
$\$A$	$aabbb$	s	$(s, a, \varepsilon) \to (s, A)$
$\$AA$	$abbb$	s	$(s, a, \varepsilon) \to (s, A)$
$\$AAA$	bbb	s	$(s, b, A) \to (p, \varepsilon)$
$\$AA$	bb	p	$(p, b, A) \to (p, \varepsilon)$
$\$A$	b	p	$(p, b, A) \to (p, \varepsilon)$
$\$$		p	$(p, \varepsilon, \$) \to (q, \varepsilon)$
$\$$		q	ACCEPTÉ !

Les tables d'analyse LL et SLR vues précédemment sont des automates à pile (les transitions ne sont pas écrites de la même manière mais ça revient au même).

Chapitre 8

Analyse sémantique

Certaines propriétés fondamentales du langage source à traduire ne peuvent être décrites à l'aide de la seule grammaire hors contexte du langage, car justement elles dépendent du contexte. Par exemple (exemples seulement, car cela dépend du langage source) :

- on ne peut pas utiliser dans une instruction une variable qui n'a pas été déclarée au préalable
- on ne peut pas déclarer deux fois une même variable
- lors d'un appel de fonction, il doit y avoir autant de paramètres formels que de paramètres effectifs, et leur type doit correspondre
- on ne peut pas multiplier un réel avec une chaîne
- ⋮

Le rôle de l'analyse sémantique (que l'on appelle aussi **analyse contextuelle**) est donc de vérifier ces contraintes. Elle se fait en général en même temps que l'analyse syntaxique, à l'aide d'actions sémantiques insérées dans les règles de productions, c'est à dire à l'aide de *définitions dirigées par la syntaxe* (DDS).

Les contraintes à vérifier dépendant fortement des langages, il n'y a pas vraiment de méthode universelle pour effectuer l'analyse sémantique. Dans ce chapitre, nous donnerons (après avoir introduit ce qu'est une DDS) une liste de problèmes rencontrés plutot que des solutions qui marchent à tous les coups.

8.1 Définition dirigée par la syntaxe

Une définition dirigée par la syntaxe (DDS) est un formalisme permettant d'associer des actions à une production d'une règle de grammaire.

• Chaque symbole de la grammaire (terminal ou non) possède un ensemble d'**attributs** (i.e. des variables).

• Chaque règle de production de la grammaire possède un ensemble de **règles sémantiques** qui permettent de calculer la valeur des attributs associés aux symboles apparaissant dans la production.

• Une règle sémantique est une suite d'instructions algorithmiques: elle peut contenir des affectations, des si-sinon, des instructions d'affichage, ...

Définition 8.1 *On appelle **définition dirigée par la syntaxe** (DDS), la donnée d'une grammaire et de son ensemble de règles sémantiques. On parle également de **grammaire attribuée**.*

On notera $X.a$ l'attribut a du symbole X. S'il y a plusieurs symboles X dans une production, on notera $X^{(0)}$ s'il est en partie gauche, $X^{(1)}$ si c'est le plus à gauche de la partie droite, $X^{(2)}$ si c'est le deuxième plus à gauche de la partie droite, ..., $X^{(n)}$ si c'est le plus à droite de la partie droite.

Un exemple :
Grammaire $S \to aSb|aS|cSacS|\varepsilon$
attributs : nba (calcul du nombre de a), nbc (du nombre de c)
une DDS permettant de calculer ces attributs pourrait être :

production	Règle sémantique
$S \to aSb$	$S^{(0)}$.nba:=$S^{(1)}$.nba+1
	$S^{(0)}$.nbc:=$S^{(1)}$.nbc
$S \to aS$	$S^{(0)}$.nba:=$S^{(1)}$.nba+1
	$S^{(0)}$.nbc:=$S^{(1)}$.nbc
$S \to cSacS$	$S^{(0)}$.nba:=$S^{(1)}$.nba+$S^{(2)}$.nba+1
	$S^{(0)}$.nbc:=$S^{(1)}$.nbc+$S^{(2)}$.nbc+2
$S \to \varepsilon$	$S^{(0)}$.nba:=0
	$S^{(0)}$.nbc:=0
$S' \to S$	// le résultat final est dans S.nba et S.nbc

Dans cet exemple de DDS, les attributs des symboles en parties gauches dépendent des attributs des symboles de la partie droite.
On peut dessiner un arbre syntaxique avec la valeur des deux attributs pour chaque symbole non terminal, afin de mieux voir/comprendre comment se fait le calcul. Par exemple pour le mot *acaacabb* :

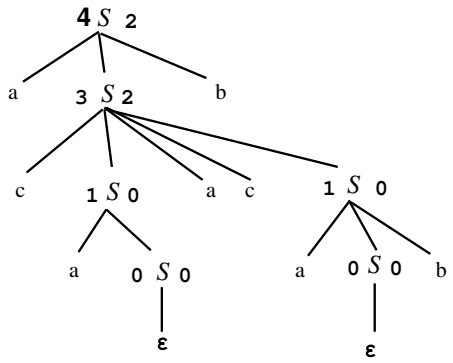


FIG. 8.1 - *Calcul du nombre de a et b pour acaacabb*

Sur cet arbre syntaxique (et donc pour cette DDS), on voit que l'on peut calculer les attributs d'un noeud dès que les attributs de tous ses fils ont été calculés.

Définition 8.2 *On appelle* ***arbre syntaxique décoré*** *un arbre syntaxique sur les noeuds duquel on rajoute la valeur de chaque attribut.*

Dans une DDS, il n'y a **aucun ordre spécifique** imposé pour l'évaluation des attributs. Tout ordre d'évaluation qui calcule sur l'arbre syntaxique l'attribut a après tous les attributs dont a dépend est possible. Plus généralement, il n'y a aucun ordre spécifique pour l'execution des actions sémantiques.

Reprenons l'exemple et étudions une autre DDS pour ce problème, une DDS qui ne ferait pas intervenir d'attributs, mais utiliserait de simples variables :

production	Règle sémantique
$S \to aSb$	nba++
$S \to aS$	nba++
$S \to cSacS$	nba++
	nbc+ = 2
$S \to \varepsilon$	

Cette DDS pose un gros problème : où initialiser les variables? Si on le fait dans l'action sémantique associée à la régle $S \to \varepsilon$, RIEN ne nous oblige à exécuter cette initialisation AVANT les actions d'incrémentation des variables. Pareil si on rajoute une production $S' \to S$.
Conclusion : cette DDS ne marche pas, il **faut** utiliser des attributs.

On distingue deux types d'attributs : les synthétisés et les hérités. Ce qui les distingue c'est la façon dont ils sont calculés.

Définition 8.3 *Un attribut est dit* ***synthétisé*** *lorsqu'il est calculé pour le non terminal de la partie gauche en fonction des attributs des non terminaux de la partie droite.*

Sur l'arbre décoré : la valeur d'un attribut en un noeud se calcule en fonction des attributs de ses **fils**. C'est à dire que le le calcul de l'attribut se fait des feuilles vers la racine.

Dans l'exemple donné, nba et nbc sont des attributs synthétisés.

Définition 8.4 *Une grammaire attribuée n'ayant que des attributs synthétisés est dite* **S-attribuée**

Les attributs synthétisés peuvent être facilement évalués lors d'une analyse ascendante (donc par exemple avec `yacc/bison`). Mais pas du tout lors d'une analyse descendante.

Définition 8.5 *Un attribut est dit* ***hérité*** *lorsqu'il est calculé à partir des attributs du non terminal de la partie gauche, et éventuellement des attributs d'autres non terminaux de la partie droite.*

Sur l'arbre décoré : la valeur d'un attribut à un noeud se calcule en fonction des attributs des **frères et du père**. C'est à dire que le calcul de l'attribut se fait de la racine vers les feuilles.

Remarque : l'axiome de la grammaire doit être de la forme $A \to B$ (un seul non-terminal) pour pouvoir initialiser le calcul.

Définition 8.6 *Une grammaire attribuée n'ayant que des attributs hérités* ***et telle que*** *ces attributs ne dépendent pas des attributs des frères* ***droits*** *est appelée* **L-attribuée**

Les attributs d'une grammaire L-attribuée peut peuvent être facilement évalués lors d'une analyse descendante.

8.1.1 Schéma de traduction dirigé par la syntaxe

Définition 8.7 *Un* schéma de traduction dirigé par la syntaxe *(STDS) est une DDS dans laquelle l'ordre d'exécution des actions sémantiques est imposé.*

Exemple :
$S' \to$ { nba := 0 } S
$S \to$ a { nba ++ } S

Dans un STDS, si l'on a :
$S \to$ α X { *une action* } Y β
l'action est exécutée après que le sous-arbre syntaxique issu de X aura été parcouru (par un parcours en profondeur) et avant que celui issu de Y ne le soit.

Remarque : `yacc/bison` permet de faire des STDS de grammaires S-attribuées.

8.1.2 Graphe de dépendances

Une DDS peut utiliser à la fois des attributs synthétisés et des attributs hérités. Il n'est alors pas forcémment évident de s'y retrouver pour calculer ces attributs. L'attribut machin dépend de l'attribut bidule qui lui-même dépend de l'attribut truc ...Il faut établir un **ordre d'évaluation** des règles sémantiques. On construira ce qu'on appelle un **graphe de dépendances**.

Exemple : Soit la DDS suivante (totalement stupide) qui fait intervenir deux attributs hérités h et r et deux attributs synthétisés s et t.

production	action sémantique
$S \to E$	E.h:=$2 * E$.s+1
	S.r:=E.t
$E \to EE$	$E^{(0)}$.s:=$E^{(1)}$.s+$E^{(2)}$.s
	$E^{(0)}$.t:=$3 * E^{(1)}$.t−$E^{(2)}$.t
	$E^{(1)}$.h:=$E^{(0)}$.h
	$E^{(2)}$.h:=$E^{(0)}$.h+4
$E \to$ nombre	E.s:= nombre
	E.t:=E.h +1

La figure 8.2 donne un exemple d'arbre décoré.

Définition 8.8 *On appelle* ***graphe de dépendances*** *le graphe orienté représentant les interdépendances entre les divers attributs. Le graphe a pour sommet chaque attribut. Il y a un arc de a à b ssi le calcul de b dépend de a.*

On construit le graphe de dépendances pour chaque règle de production, ou bien directement le graphe de dépendances d'un arbre syntaxique donné. C'est ce dernier qui permet de voir dans quel ordre évaluer les attributs. Mais il est toujours utile d'avoir le graphe de dépendances pour chaque règle de production afin d'obtenir "facilement" le graphe pour n'importe quel arbre syntaxique.

Sur l'exemple : La figure 8.3 donne le graphe de dépendances pour chaque règle de production, et la figure 8.4 le graphe de dépendance de l'arbre syntaxique.

Remarque : si le graphe de dépendance contient un cycle, l'évaluation des attributs est alors **impossible**.

Exemple : Soit la DDS suivante (a hérité et b synthétisé)

production	action sémantique
$S' \to S$	S.a:=S.b
$S \to SST$	$S^{(0)}$.b:=$S^{(1)}$.b+$S^{(2)}$.b+T.b
	$S^{(1)}$.a:=$S^{(0)}$.a
	$S^{(2)}$.a:=$S^{(0)}$.a+1
	T.a:=$S^{(0)}$.a+$S^{(0)}$.b+$S^{(1)}$.a
$T \to PT$	$T^{(0)}$.b:=P.a+P.b
	P.a:=$T^{(0)}$.a+3
	$T^{(1)}$.a:=...
⋮	

Le graphe de dépendance associé peut contenir un cycle (voir figure 8.5), le calcul est donc impossible.

8.1.3 Evaluation des attributs

Après l'analyse syntaxique

On peut faire le calcul des attributs indépendamment de l'analyse syntaxique : lors de l'analyse syntaxique, on construit (en dur) l'arbre syntaxique[1], puis ensuite, lorsque l'analyse syntaxique est terminée, le calcul des attributs s'effectue sur cet arbre par des parcours de cet arbre (avec des aller-retours dans l'arbre suivant l'ordre d'évaluation des attributs).
Cette méthode est très couteuse en mémoire (stockage de l'arbre). Mais l'avantage est que l'on n'est pas dépendant de l'ordre de visite des sommets de l'arbre syntaxique imposé par l'analyse syntaxique (une analyse descendante impose un parcours en profondeur du haut vers le bas, de la gauche vers la droite, une analyse ascendante un parcours du bas vers le haut, ...).
On peut également construire l'arbre en dur pour une sous-partie seulement du langage.

Pendant l'analyse syntaxique

On peut évaluer les attributs en même tant que l'on effectue l'analyse syntaxique. Dans ce cas, on utilisera une pile pour conserver les valeurs des attributs, cette pile pouvant être la même que celle de l'analyseur syntaxique, ou une autre.
Cette fois-ci, l'ordre d'évaluation des attributs est tributaire de l'ordre dans lequel les noeuds de l'arbre syntaxique sont "crées" [2] par la méthode d'analyse. C'est à dire que l'on ne pourra traiter les grammaires S-attribuées qu'avec une analyse ascendante, et les grammaires L-attribuées qu'avec une analyse descendante. Pour les grammaires ni L-attribuées ni S-attribuées, on est emmerdé.

• Exemple avec un attribut synthétisé : évaluation d'une expression arithmétique avec une analyse ascendante (ce que fait `yacc/bison`). Une DDS (un STDS plutôt, pour ce qui concerne les manip de la pile, qui doivent être faites en fin de chaque production) est

1. en fait, on ne construit pas exactement l'arbre syntaxique, mais un **arbre abstrait** qui est une forme condensée de l'arbre syntaxique
2. attention, l'arbre n'est pas réellement construit

production	action sémantique	traduction avec une pile
$E \to E + T$	$E^{(0)}$.val:=$E^{(1)}$.val+T.val	tmpT = depiler() tmpE = depiler() empiler(tmpE+tmpT)
$E \to E - T$	$E^{(0)}$.val:=$E^{(1)}$.val−T.val	tmpT = depiler() tmpE = depiler() empiler(tmpE−tmpT)
$E \to T$	E.val:=T.val	
$T \to T * F$	$T^{(0)}$.val:=$T^{(1)}$.val∗F.val	tmpF = depiler() tmpT = depiler() empiler(tmpT∗tmpF)
$T \to F$	T.val:=F.val	
$F \to (E)$	F.val:=E.val	
$F \to$ nb	F.val:=nb	empiler(nb)

L'analyse syntaxique s'effectue à partir de la table d'analyse LR donnée page 31. Par exemple, pour le mot $2 * (10 + 3)$ on obtiendra :

pile	entrée	action	pile des attributs
\$ 0	$2 * (10 + 3)$\$	d5	
\$ 0 2 5	$*(10 + 3)$\$	r6 : $F \to$ nb	2
\$ 0 F 3	$*(10 + 3)$\$	r4 : $T \to F$	2
\$ 0 T 2	$*(10 + 3)$\$	d7	2
\$ 0 T 2 ∗ 7	$(10 + 3)$\$	d4	2
\$ 0 T 2 ∗ 7 (4	$10 + 3)$\$	d5	2
\$ 0 T 2 ∗ 7 (4 10 5	$+3)$\$	r6 : $F \to$ nb	2 10
\$ 0 T 2 ∗ 7 (4 F 3	$+3)$\$	r4 : $T \to F$	2 10
\$ 0 T 2 ∗ 7 (4 T 2	$+3)$\$	r2 : $E \to T$	2 10
\$ 0 T 2 ∗ 7 (4 E 8	$+3)$\$	d6	2 10
\$ 0 T 2 ∗ 7 (4 E 8 + 6	$3)$\$	d5	2 10
\$ 0 T 2 ∗ 7 (4 E 8 + 6 3 5	$)$\$	r6 : $F \to$ nb	2 10 3
\$ 0 T 2 ∗ 7 (4 E 8 + 6 F 3	$)$\$	r4 : $T \to F$	2 10 3
\$ 0 T 2 ∗ 7 (4 E 8 + 6 T 9	$)$\$	r1 : $E \to E + T$	2 13
\$ 0 T 2 ∗ 7 (4 E 8	$)$\$	d11	2 13
\$ 0 T 2 ∗ 7 (4 E 8) 11	\$	r5 : $F \to (E)$	2 13
\$ 0 T 2 ∗ 7 F 10	\$	r3 : $T \to T * F$	26
\$ 0 T 2	\$	r2 : $E \to T$	26
\$ 0 E 1	\$	**ACCEPTÉ !!!**	26

• Exemple avec un attribut hérité : calcul du niveau d'imbrication des) dans un système de parenthèses bien formé.

production	action sémantique	traduction avec une pile
$S' \to S$	S.nb:=0	empiler(0)
$S \to (S)S$	$S^{(1)}$.nb:=$S^{(0)}$.nb+1 $S^{(2)}$.nb:=$S^{(0)}$.nb	tmp=depiler() empiler(tmp) empiler(tmp+1) } empiler(sommet()+1)
$S \to \varepsilon$	écrire S.nb	écrire depiler()

On effectue une analyse descendante, donc il nous faut la table d'analyse LL : PREMIER(S') =PREMIER(S) = $\{(, \varepsilon\}$, SUIVANT(S') = {\$} et SUIVANT(S) = $\{), \$\}$, donc

	(	)	\$
S'	$S' \to S$		$S' \to S$
S	$S \to (S)S$	$S \to \varepsilon$	$S \to \varepsilon$

Analysons le mot (()(()))()

pile	entrée	action	pile des attributs	écritures
\$ S'	(()(()))()\$	$S' \to S$	0	
\$ S	(()(()))()\$	$S \to (S)S$	0 1	
\$ $S)S($	(()(()))()\$		0 1	
\$ $S)S$	()(()))()\$	$S \to (S)S$	0 1 2	
\$ $S)S)S($	()(()))()\$		0 1 2	
\$ $S)S)S$	)(()))()\$	$S \to \varepsilon$	0 1	2
\$ $S)S)$	)(()))()\$		0 1	
\$ $S)S$	(()))()\$	$S \to (S)S$	0 1 2	
\$ $S)S)S($	(()))()\$		0 1 2	
\$ $S)S)S$	()))()\$	$S \to (S)S$	0 1 2 3	
\$ $S)S)S)S($	()))()\$		0 1 2 3	
\$ $S)S)S)S$	)))()\$	$S \to \varepsilon$	0 1 2	3
\$ $S)S)S)$	)))()\$		0 1 2	
\$ $S)S)S$	))()\$	$S \to \varepsilon$	0 1	2
\$ $S)S)$	))()\$		0 1	
\$ $S)S$	)()\$	$S \to \varepsilon$	0	1
\$ $S)$	)()\$		0	
\$ S	()\$	$S \to (S)S$	0 1	
\$ $S)S($	()\$		0 1	
\$ $S)S$	)\$	$S \to \varepsilon$	0	1
\$ $S)$	)\$		0	
\$ S	\$	$S \to \varepsilon$		0
\$	\$			
		FINI		

• Lorsque l'on a à la fois des attributs synthétisés et hérités, on peut essayer de remplacer les attributs hérités par des synthétisés (qui font la même chose of course).
Exemple : compter le nombre de bit à 1 dans un mot binaire.

Grammaire : $\begin{cases} S \to B \\ B \to 0B|1B|\varepsilon \end{cases}$

DDS avec attribut hérité :

production	action sémantique
$S \to B$	B.nb:=0
$B \to 0B$	$B^{(1)}$.nb:=$B^{(0)}$.nb
$B \to 1B$	$B^{(1)}$.nb:=$B^{(0)}$.nb+1
$B \to \varepsilon$	écrire B.nb

DDS avec attribut synthétisé :

production	action sémantique
$S \to B$	écrire B.nb
$B \to 0B$	$B^{(0)}$.nb:=$B^{(1)}$.nb
$B \to 1B$	$B^{(0)}$.nb:=$B^{(1)}$.nb+1
$B \to \varepsilon$	B.nb:=0

La figure 8.6 donne les arbres décorés pour le mot 10011.

• Parfois, il est nécessaire de modifier la grammaire. Par exemple, considérons une DDS de typage de variables dans une déclaration Pascal de variables (en Pascal, une déclaration de variable consiste en une liste d'identificateurs séparés par des virgules, suivie du caractère ':', suivie du type. Par exemple : `a,b : integer`)

production	action sémantique
$D \to L$: T	L.type:=T.type
$L \to$ Id	mettre dans la table des symboles le type L.type pour Id
$L \to L$, Id	$L^{(1)}$.type:=$L^{(0)}$.type mettre dans la table des symboles le type $L^{(0)}$.type pour Id
$T \to$ `integer`	T.type:=entier
$T \to$ `char`	T.type:=caractere

C'est un attribut hérité. Pas possible de trouver un attribut synthétisé faisant la même chose avec cette grammaire. Changeons la grammaire !

production	action sémantique
$D \rightarrow$ Id L	mettre dans la table des symboles le type L.type pour Id
$L \rightarrow$, Id L	$L^{(0)}$.type:=$L^{(1)}$.type mettre dans la table des symboles le type $L^{(1)}$.type pour Id
$L \rightarrow$: T	L.type:=T.type
$T \rightarrow$ `integer`	T.type:=entier
$T \rightarrow$ `char`	T.type:=caractere

Et voilà !

• Mais bon, ce n'est pas toujours évident de concevoir une DDS n'utilisant que des attributs hérités ou de concevoir une autre grammaire permettant de n'avoir que des synthétisés. Heureusement, il existe des méthodes automatiques qui transforment une DDS avec des attributs hérités et synthétisés en une DDS équivalente n'ayant que des attributs synthétisés. Mais nous n'en parlerons pas ici, cf bouquins.

• Une autre idée peut être de faire plusieurs passes d'analyses, suivant le graphe de dépendances de la DDS. Par exemple (DDS donnée page 53) : une passe ascendante permettant d'évaluer un certain attribut synthétisé s, puis une passe descendante pour évaluer des attributs hérités h et r, puis enfin un passe ascendante pour évaluer un attribut synthétisé t.

8.2 Portée des identificateurs

On appelle *portée* d'un identificateur la(es) partie(s) du programme où il est valide et a la signification qui lui a été donné lors de sa déclaration.

Cette notion de validité et visibilité dépend bien sûr des langages. Par exemple, en Cobol tous les identificateurs sont partout visibles. En Fortran, C, Pascal, ..., les identificateurs qui sont définis dans un bloc ne sont visibles qu'à l'intérieur de ce bloc; un identificateur déclaré dans un sous-bloc masque un identificateur de même nom déclaré dans un bloc de niveau inférieur. En Algol, Pascal, Ada et dans les langages fonctionnels, il peut y avoir une imbrication récursive des blocs sur une profondeur quelconque (cf `letrec`)...

Par exemple, l'analyseur sémantique doit vérifier si chaque utilisation d'un identificateur est légale, ie si cet identificateur a été déclaré et cela de manière unique dans son bloc. Il faut donc mémoriser tous les symboles rencontrés au fur et à mesure de l'avancée dans le texte source. Pour cela on utilise une structure de données adéquate que l'on appelle une **table des symboles**.
La table des symboles contient toutes les informations nécessaires sur les symboles (variables, fonctions, ...) du programme :

- identificateur (le nom du symbole dans le programme)
- son type (entier, chaîne, fonction qui retourne un réel,...)
- son emplacement mémoire
- si c'est une fonction : la liste de ses paramètres et leurs types
- ...

Lors de la **déclaration**[3] d'une variable, elle est stockée dans cette table (avec les informations que l'on peut déja connaitre). Il faut préalablement regarder si cette variable n'est pas déjà contenue dans la table. Si c'est le cas, c'est une erreur. Lors de l'**utilisation** d'une variable, il faut vérifier qu'elle fait partie de la table (on peut alors récupérer son type, donner ou modifier une valeur, ...).
Il faut donc munir la table des symboles de procédures d'ajout, suppression et recherche. La structure d'une table des symboles peut aller du plus simple au plus compliqué (simple tableau trié ou non trié, liste linéaire, forme arborescente, table d'adressage dispersé (avec donc des fonctions de hachage), ...), cela dépend de la complexité du langage à compiler, de la rapidité que l'on souhaite pour le compilateur, de l'humeur du concepteur du compilateur, ...

Dans certains langages, on peut autoriser qu'un identificateur ne soit pas déclaré dans son bloc B à condition qu'il ait été déclaré dans un bloc d'un niveau inférieur qui contient B. Il faut également gérer le cas où un identificateur est déclaré dans au moins deux blocs différents B contenu dans B' (dans ce cas, l'identificateur de plus haut niveau cache les autres).
On peut alors utiliser une **pile** pour empiler la table des symboles et gérer ainsi la portée des identificateurs. Mais il y a d'autres méthodes, comme par exemple coder la portée dans la table des symboles, pour chaque symbole.

3. et encore, certains langages n'imposent pas de déclarer les variables. Par exemple en Fortran (qui est pourtant un langage typé), tout nom de variable commençant par `I` est supposé dénoter un entier (sauf indication contraire)

8.3 Contrôle de type

Lorsque le langage source est un langage typé, il faut vérifier la pertinence des types des objets manipulés dans les phrases du langage.
Exemples : en C, on ne peut pas additionner un `double` et un `char *`, multiplier un `int` et un `struct`, indicer un tableau avec un `float`, affecter un `struct *` à un `char` ... Certaines opérations sont par contre possibles : affecter un `int` à un `double`, un `char` à un `int`, ... (conversions implicites).

Définition 8.9 *On appelle* **statique** *un contrôle de type effectué lors de la compilation, et* **dynamique** *un controle de type effectué lors de l'exécution du programme cible.*

Le contrôle dynamique est à éviter car il est alors très difficile pour le programmeur de voir d'où vient l'erreur (quand il y en a une). En pratique cependant, certains contrôles ne peuvent être fait que dynamiquement. Par exemple, si l'on a

```
int tab[10];
int i;
...
... tab[i] ...
```

le compilateur ne pourra pas garantir en général[4] qu'à l'exécution la valeur de `i` sera comprise entre 0 et 9.

Exemples de TDS de contrôle de type :

Règle de production	action sémantique	
$I \to Id = E$	I.type :=	si Id.type=E.type alors vide sinon erreur_type_incompatible(ligne,Id.type,E.type)
$I \to$ si E alors I	$I^{(0)}$.type :=	si E.type=booleen alors $I^{(1)}$.type sinon erreur_type(ligne,...)
$E \to Id$	E.type :=	Recherche_Table(Id)
$E \to E + E$	$E^{(0)}$.type :=	si $E^{(1)}$.type=entier et $E^{(2)}$.type=entier alors entier sinon si ($E^{(1)}$.type≠reel et $E^{(1)}$.type≠entier) ou ($E^{(2)}$.type≠reel et $E^{(2)}$.type≠entier) alors erreur_type_incompatible(ligne,$E^{(1)}$.type,$E^{(2)}$.type) sinon reel
$E \to E$ mod E	$E^{(0)}$.type :=	si $E^{(1)}$.type=entier et $E^{(2)}$.type=entier alors entier sinon erreur_type(ligne,...)
	⋮	

Ça a l'air assez simple, non?
Maintenant imaginons le boulot du compilateur C lorsqu'il se trouve confronté à un truc du genre

```
s->t.f(p[*i])-&j
```

Il faut alors trouver une représentation pratique pour les expressions de type.

Exemple : codage des expressions de type par Ritchie et Johnson pour un compilateur C.
On considère les constructions de types suivantes :

pointeur(t) : un pointeur vers le type t
fretourne(t) : une fonction (dont les arguments ne sont pas spécifiés) qui retourne un objet de type t
tableau(t) : un tableau (de taille indeterminée) d'objets de type `t`

Ces constructeurs étant unaires, les expressions de type formés en les appliquant à des types de base ont une structure très uniforme. Par exemple

caractère
fretourne(caratère)
tableau(caractère)
fretourne(pointeur(entier))
tableau(pointeur(fretourne(entier)))

Chaque constructeur peut être représentée par une séquence de bits selon un procédé d'encodage simple, de même que chaque type de base. Par exemple

constructeur	codage	type de base	codage
pointeur	01	*entier*	0000
tableau	10	*booléen*	0001
fretourne	11	*caractère*	0010
		réél	0011

4. quoi qu'il existe des techniques d'analyse de flot de données qui permettent de résoudre ce problème dans certains cas particuliers

Les expressions de type peuvent maintenant être encodée comme des séquences de bit

expression de type	codage
caractère	000000 0010
fretourne(caractère)	000011 0010
pointeur(fretourne(caractère))	000111 0010
tableau(pointeur(fretourne(caractère)))	100111 0010

Cette méthode a au moins deux avantages: économie de place et il y a une trace des différents constructeurs appliqués.

8.3.1 Surcharge d'opérateurs et de fonctions

Des opérateurs (ou des fonctions) peuvent être **surchargés** c'est à dire avoir des significations différentes suivant le contexte. Par exemple, en C, la division est la division entière si les deux opérandes sont de type entier, et la division réelle sinon. Dans certains langages[5], on peut redéfinir un opérateur standard, par exemple en Ada les instructions

```
function "*" (i,j : integer) return integer;
function "*" (x,y : complexe) return complexe;
```

surchargent l'opérateur `*` pour effectuer aussi la multiplication de nombres complexes (le type `complexe` devant être défini). Maintenant, si l'on rencontre `3*5`, doit-on considérer la multiplication standard qui retourne un entier ou celle qui retourne un complexe? Pour répondre à cette question, il faut regarder comment est utilisée l'expression. Si c'est dans `2+(3*5)` c'est un entier, si c'est dans `z*(3*5)` où `z` est un complexe, alors c'est un complexe. Bref, au lieu de remonter un seul type dans la TDS, il faudra remonter un ensemble de types possibles, jusqu'à ce que l'on puisse conclure. Et je ne parle pas de la génération de code ...

8.3.2 Fonctions polymorphes

Une fonction (ou un opérateur) est dite *polymorphe* lorsque l'on peut l'appliquer à des arguments de types variables (et non fixés à l'avance). Par exemple, l'opérateur `&` en C est polymorphe puisqu'il s'applique aussi bien à un entier qu'à un caractère, une structure définie par le programmeur, ... On peut également, dans certains langages, se définir une fonction calculant la longueur d'une liste d'éléments, quelque soit le type de ces éléments. Cette fois-ci, il ne s'agit plus seulement de vérifier l'équivalence de deux types, mais il faut les **unifier**.
Un des problème qui se pose alors est **l'inférence de type**, c'est à dire la détermination du type d'une construction du langage à partir de la façon dont elle est utilisée.
Ces problèmes ne sont pas simples et sont étudiés dans le cadre des recherches en logique combinatoire et en lambda-calcul. Le lambda-calcul permet de définir et d'appliquer les fonctions sans s'occuper des types.

8.4 Conclusion

L'introduction de vérifications contextuelles dans l'analyseur occupe une place très importante et est extrémement délicate[6].

5. par exemple dans les langages objets
6. Et en plus personnellement je trouve ça très chiant

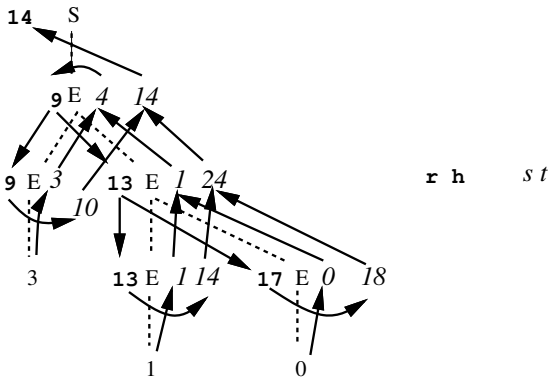


FIG. 8.2 - *Attributs hérités et synthétisés*

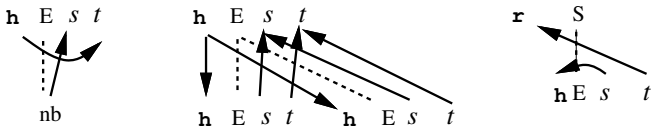


FIG. 8.3 - *Graphes de dépendances pour les productions*

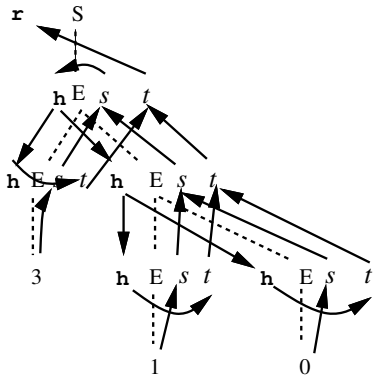


FIG. 8.4 - *Graphe de dépendances de l'arbre syntaxique*

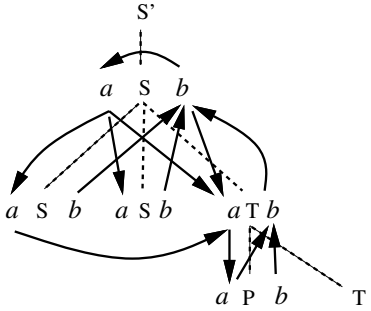


FIG. 8.5 - *Cycle dans le graphe de dépendance*

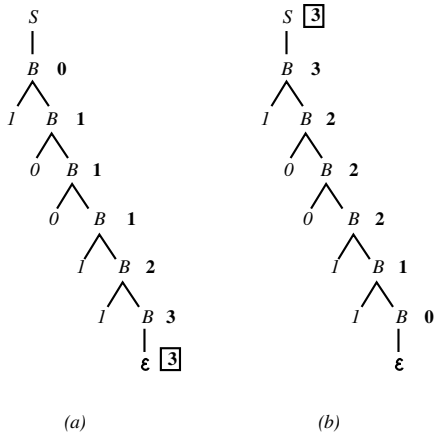


FIG. 8.6 - *Arbre décoré pour* 10011 *avec des attributs (a)hérités (b) synthétisés*

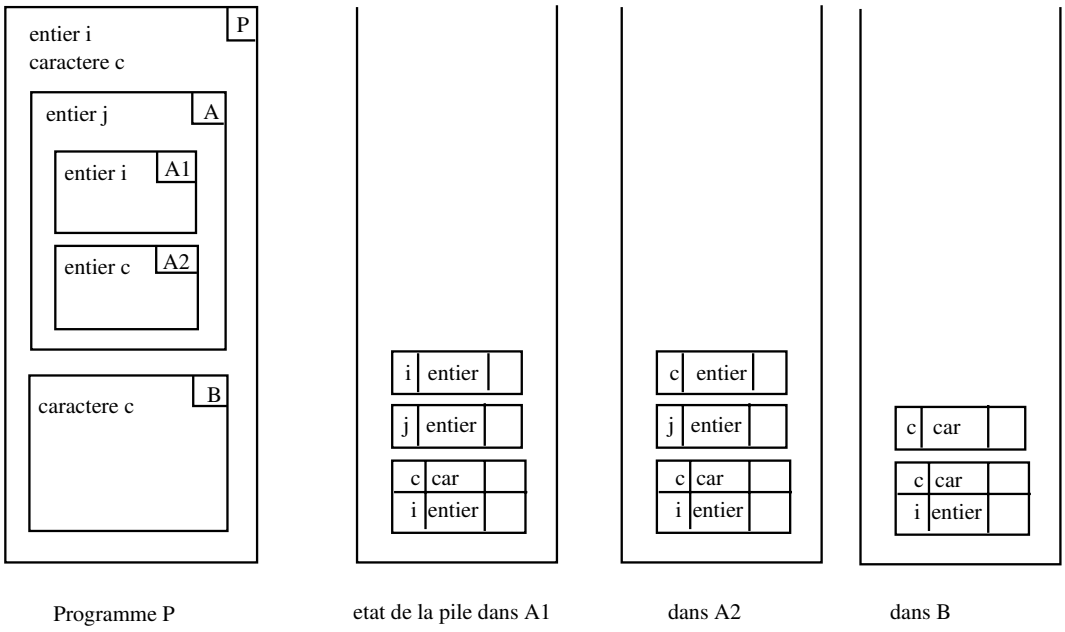


FIG. 8.7 - *Portée des identificateurs : empilement de la table des symboles*

Chapitre 9

Génération de code

Avant d'attaquer le problème de la génération du code cible, il faut se poser des questions sur l'organisation de la mémoire (l'environnement d'exécution) :
- gestion du flot de contrôle (appels et retour de fonctions, instructions de saut (du type `break` en C))
- stockage des variables
- allocation dynamique de mémoire
- etc

Ensuite, il s'agit de produire le code cible.
Bien qu'un texte source puisse être traduit directement en langage cible, on utilise en général une forme intermédiaire indépendante de la machine cible. On produira donc dans un premier temps du **code intermédiaire**, que l'on pourra ensuite **optimiser**, puis enfin à partir du code intermédiaire on produira le code cible.

9.1 Environnement d'exécution

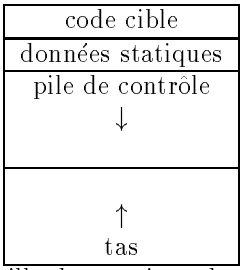
9.1.1 Organisation de la mémoire à l'exécution

Questions liées au langage source :
- les fonctions peuvent-elles être récursives?
- une fonction peut-elle faire référence à des noms non locaux?
- comment sont passés les paramètres lors d'un appel de fonction?
- peut-on allouer dynamiquement de la mémoire?
- doit-on libérer explicitement l'espace mémoire alloué dynamiquement?
- que deviennent les variables locales à une fonction lorsque l'on sort de la fonction?
- ...

Questions liées à la machine cible :
- quelle est la longueur d'un mot ou d'une adresse?
- quelles sont les entités directement adressables de la machine?
- existe-t-il des instructions pour accéder directement et efficacement à des parties d'entités directement adressables?
- est-ce possible de rassembler plusieurs "petits" objets (booléen, caractère, ...) dans des unités plus grandes?
- y-a-t-il des restrictions d'adressage (conditions d'alignements)?
- ...

Les réponses à ces questions influent sur la façon d'organiser la mémoire.

(modèle général)
On divise le bloc de mémoire allouées à l'exécution par le système d'exploitation cible en 4 zones :



• données statiques : la taille de certaines données est connue dès la compilation. On les place alors dans une zone déterminée statiquement. L'avantage c'est que l'on peut alors compiler directement leur adresse dans le

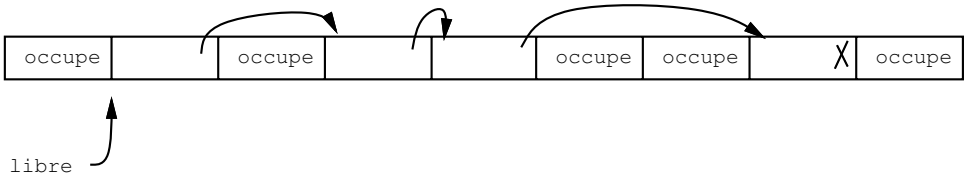
code cible.
• pile de contrôle : elle permet de gérer les appels de fonctions. On y stocke les *enregistrements d'activation*, c'est à dire toutes les informations nécessaires lors de l'activation et du retour d'une fonction (adresse de retour, valeur de certains registres, ...). Lors d'un appel de fonction, un *protocole d'appel* alloue un enregistrement d'activation, remplit ses champs et l'empile. Lors du retour, la *séquence de retour* dépile et restaure l'état de la machine afin de permettre à la fonction appelante de continuer l'exécution.
• tas : contient tout le reste, c'est à dire les variables allouées dynamiquement au cours de l'exécution du programme.

9.1.2 Allocation dynamique : gestion du tas

Les allocations peuvent être explicites (instructions `new` en Pascal, `malloc` en C, ...) ou implicites (utilisation de `cons` en Lisp, appels implicites aux constructeurs en C++, ...). Les données allouées sont conservées jusquà leur libération explicite (instructions `free` en C, `dispose` en Pascal, ...) ou implicite (appels implicites des destructeurs en C++, *garbage collector* en Lisp, ...).

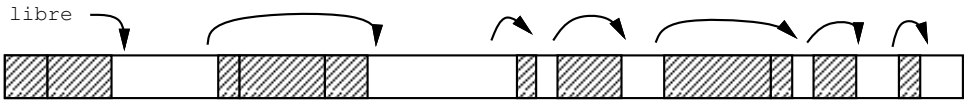
- **Allocation explicite de blocs de taille fixe**

  gestion des blocs libre par une liste



- **Allocation explicite de blocs de taille variable**

  Même principe avec des infos en plus dans chaque bloc alloué (taille du bloc, ...)



  Problème : si l'utilisateur demande à allouer un bloc de taille superieure au plus grand des blocs libres.

- **Libération implicite**

  Intéret : évite les *rebuts* (ou *miettes*), c'est à dire les zones mémoires allouées mais devenues innaccessibles, et les *références fantômes*, c'est à dire une référence à une zone qui a été libérée.
  Modèle de format de bloc :

taille du bloc
compteur de référence et/ou marquage
pointeurs vers des blocs
informations utilisateurs

  Il y a deux méthodes pour gérer la libération implicite :

  - **Compteur de références** : On suppose qu'un bloc est utilisé s'il est possible au programme utilisateur de faire référence à l'info contenue dedans (par un pointeur ou en suivant une liste de pointeurs). On incrémente alors le compteur de référence à chaque nouvelle référence à ce bloc, et on le décrémente à chaque "disparition" de référence (sortie de fonction dans laquelle la référence est locale par exemple). Lorsque le compteur devient nul, on peut libérer le bloc.
    Exemple : si on a une instruction (C) `p=q`, le compteur du bloc référencé par `p` est décrémenté, celui de `q` est incrémenté.
    Attention, cette méthode merde si on a des blocs qui forment une liste circulaire.
  - **Marquage** : L'autre approche consiste à suspendre temporairement l'exécution du programme et faire du *ramasse-miettes*. Cela necessite que tous les pointeurs vers le tas soient connus. On commence par marquer tout le tas en tant qu'inutilisé. Puis on passe en revue tous les pointeurs en cours en marquant utilisé tout bloc atteint.

- **Compression**

  De temps à autres, on peut suspendre temporairement l'exécution du programme pour faire de la compression, c'est à dire déplacer les blocs utilisés vers une extrémité du tas, afin de rassembler toute la mémoire

libre en un seul grand bloc[1]. Il ne faut pas oublier de mettre à jour toutes les références des blocs que l'on déplace[2].

9.2 Code intermédiaire

Le langage intermédiaire doit être facile à produire et facile à traduire dans le langage cible (quelqu'il soit). Pourquoi passer par un langage intermédiaire?

- le "reciblage" est facilité. En effet, le langage cible dépend énormemment de la machine sur laquelle il va tourner. Utiliser un langage intermédiaire permet de porter plus rapidement le compilateur, puisque les 9/10 du boulot sont déja fait lorsqu'on est arrivé à produire du code intermédiaire.
- l'optimisation du code est facilité. Il existe de nombreux algorithmes et méthodes d'optimisation de code pour les codes intermédiaires les plus utilisés.

9.2.1 Caractéristiques communes aux machines cibles

(cf cours Archi/Système ...)

- hiérarchie de mémoire : registres du processeur, mémoire principale, mémoire cache, mémoire auxiliaire (les deux dernières, nous, en compil, on s'en fout)

- les registres : universels, flottants, de données et d'adresse (parfois), de base et d'index (parfois), compteur d'instruction.
  L'accès aux registres est plus rapide que l'accès à la mémoire principale, donc un bon compilateur est un compilateur qui utilise le plus possible les registres pour les résultats intermédiaires, les calculs d'adresse, la transmission des paramètres d'une fonction, le retour des valeurs d'une fonction, ...
  Bien sûr, le nombre de registres est limité. Un problème important de la génération de code sera donc **l'allocation des registres**.

- jeu d'instructions :
  - instructions de calcul (arithmétique, booléen, ...), souvent différenciées selon la longueur des opérandes
  - instructions de transfert (entre et dans la mémoire et les registres)
  - instructions de branchements
  - instructions de communication avec le sytème d'exploitation et les périphériques (entrées/sorties)

9.2.2 Code à 3 adresses simplifié

Nous allons voir un code intermédiaire souvent utilisé : le code à 3 adresses.

Un programme écrit en code à 3 adresses est une séquences d'instructions **numérotées**, ces instructions pouvant être

affectation binaire	`(num) x := y op z`
affectation unaire	`(num) x := opu z`
affectation indicée	`(num) y[i]:= z`
copie	`(num) x:=y`
branchement inconditionnel	`(num1) aller a (num2)`
branchement conditionnel	`(num1) si x oprel y aller a (num2)`
lecture	`(num) lire x`
ecriture	`(num) ecrire y`

avec

`op`	opérateur binaire comme $+, *, -, /$, ET , OU, $\geq$, $=$...
`opu`	opérateur unaire comme $-$, $\sqrt{}$, NON, ...
`oprel`	opérateur relationnel $(=, <, \geq, \neq, \ldots)$
`x`	adresse[3] de variable ou registre
`y`, `z`	adresse de variable, registre ou constante

et **c'est tout**.
Le nombre de registres disponibles est **illimité**

Exemple 1 : le fragment de programme C `a=b*c+b*(b-c)*(-c)` devient

1. dans le cas d'une allocation de blocs de taille fixe ça n'a pas grand intérêt
2. je vous laisse imaginer le bordel pour les listes de blocs

```
(1) t1 := b*c          (5) t5 := t3*t4
(2) t2 := b-c          (6) t6 := t1+t5
(3) t3 := b*t2         (7) a := t6
(4) t4 := -c
```

ou encore, en optimisant le nombre de registres utilisés

```
(1) t1 := b*c          (5) t2 := t3*t4
(2) t2 := b-c          (6) t1 := t1+t5
(3) t2 := b*t2         (7) a := t1
(4) t3 := -c
```

Exemple 2 : soit le fragment de code C suivant

```
z=0;
do {
  x++;
  if (y<10)
    {
       z=z+2*x;
       h=5;
    }
  else
       h=-5;
  y+=h;
} while (x<=50 && z<=5*y);
```

Le code à 3 adresses correspondant est

```
(1) t1 := 0                    (12) h := t6                          (23) .....
(2) z := t1                    (13) aller a (16)
(3) t2 := x + 1                (14) t7 := -5
(4) x := t2                    (15) h := t7
(5) t3 := y<10                 (16) t8 := y+h
(6) si t3=vrai aller a (8)     (17) y := t8
(7) aller a 14                 (18) t9 := x<=50
(8) t4 := 2*x                  (19) t10 := 5*y
(9) t5 := z+t4                 (20) t11 := z<=t10
(10) z := t5                   (21) t12 := t9 et t11
(11) t6 := 5                   (22) si t12=vrai aller a (3)
```

Pour la condition du `while`, on pourrait ne pas tout évaluer et conclure avant (si `t9` est faux, inutile (et dangereux?) de calculer `t10`, `t11` et `t12`), c'est à dire qu'au lieu de faire de l'évaluation `paresseuse`, ça serait mieux de faire de l'évaluation **court-circuit** :

```
(1) t1 := 0                    (12) h := t6                (23) si t11=vrai aller (3)
(2) z := t1                    (13) aller a (16)           (24) ....
(3) t2 := x + 1                (14) t7 := -5
(4) x := t2                    (15) h := t7
(5) t3 := y<10                 (16) t8 := y+h
(6) si t3=vrai aller a (8)     (17) y := t8
(7) aller a 14                 (18) t9 := x<=50
(8) t4 := 2*x                  (19) si t9=vrai aller (21)
(9) t5 := z+t4                 (20) aller (24)
(10) z := t5                   (21) t10 := 5*y
(11) t6 := 5                   (22) t11 := z<=t10
```

9.2.3 Production de code à 3 adresses

Il est (assez) simple d'insérer des actions sémantiques dans la grammaire du langage afin de produire ce code. On utilise donc une DDS.

Expressions arithmétiques

Exemple 1 : génération de code pour les expressions arithmétiques. Soit la grammaire

$$\begin{cases} I \rightarrow Id = E \\ E \rightarrow E + E | E * E | - E | (E) | Id \end{cases}$$

La DDS utilise deux attributs : *code* (pour le ...code !) et *place* (pour le nom du registre dans lequel on range le calcul), cf figure 9.1

Production	Règle sémantique	
$I \rightarrow Id = E$	I.code=	E.code
		Id.place ":=" E.place
$E \rightarrow Id$	E.place=	Id.place
	E.code=	" "
$E \rightarrow (E)$	$E^{(0)}$.place=	$E^{(1)}$.place
	$E^{(0)}$.code=	$E^{(1)}$.code
$E \rightarrow E + E$	$E^{(0)}$.place=	NouvRegistre()
	$E^{(0)}$.code=	$E^{(1)}$.code
		$E^{(2)}$.code
		$E^{(0)}$.place ":="$E^{(1)}$.place "+" $E^{(2)}$.place
⋮	etc.	

FIG. 9.1 - *DDS de génération de code 3 adresses pour les expressions arithmétiques*

Il faudrait aussi penser aux numéros des instructions, ce n'est pas fait dans cet exemple. Mais en fait, les seuls numéros importants sont lorsqu'il y a des branchements, le reste du temps on s'en fout complètement.

Expressions booléennes

Soit la grammaire :

$$\begin{cases} I \rightarrow \text{si } L \text{ alors } I \text{ sinon } I & \\ L \rightarrow Id\ oprel\ Id & L \rightarrow \text{ non } L \\ L \rightarrow L \text{ ou } L & L \rightarrow \text{ vrai} \\ L \rightarrow L \text{ et } L & L \rightarrow \text{ faux} \end{cases}$$

Définition 9.1 *On appelle* ***évaluation paresseuse*** *l'évaluation de tous les termes qui composent une expression booléenne*

Définition 9.2 *On appelle* ***évaluation court-circuit*** *l'évaluation d'une expression booléenne qui n'évalue que les termes nécessaires*

Pour une instruction, on considère un attribut *suivant* qui contient l'étiquette de l'instruction suivante à effectuer (ce n'est pas toujours celle qui suit dans le code car il peut y avoir des branchements).

La DDS qui produit le code **paresseux** est donné figure 9.2. Cette DDS traduit le fragment de code `si a<b ou c<d et e>f alors ... sinon` en (attention à construire l'arbre de dérivation qui tient compte des priorités des opérateurs !) :

```
        t1 := a<b
        t2 := c<d
        t3 := e>f
        t4 := t2 et t4
        t5 := t1 ou t4
        si t5 aller a (Lvrai)
        aller a (Lfaux)
(Lvrai) ......
        ..... (le code du bloc alors)
        aller a (Isuiv)
(Lfaux) ......
        ..... (le code du bloc sinon)
(Isuiv) ......
```

Production	Règle sémantique
$I \rightarrow$ si L alors I sinon I	$I^{(1)}$.suivant= $I^{(0)}$.suivant $I^{(2)}$.suivant= $I^{(0)}$.suivant Vrai=NouvEtiquette() Faux=NouvEtiquette() $I^{(0)}$.code = L.code "si" L.place"=vrai aller a" Vrai "aller a" Faux "(" Vrai ")" $I^{(1)}$.code "aller a" $I^{(1)}$.suivant "(" Faux ")" $I^{(2)}$.code
$L \rightarrow L$ ou L	$L^{(0)}$.place = NouvRegistre() $L^{(0)}$.code = $L^{(1)}$.code $L^{(2)}$.code $L^{(0)}$.place ":=" $L^{(1)}$.place "ou" $L^{(2)}$.place
$L \rightarrow Id$ *oprel* Id	L.place = NouvRegistre() L.code = L.place ":=" Id.place *oprel*.valeur Id.place
$L \rightarrow$ non L	$L^{(0)}$.place = NouvRegistre() $L^{(0)}$.code = $L^{(0)}$.place ":= non" $L^{(1)}$.place
⋮	etc.

FIG. 9.2 - *DDS de génération de code 3 adresses pour l'évaluation paresseuse des expressions booléennes*

Un des problèmes qui se pose est qu'on ne peut pas connaître toutes les étiquettes destination de toutes les instructions de branchement en une seule passe. On peut résoudre le problème en faisant deux passes, ou bien encore le contourner par la technique de *reprise arrière*. On produit une série de branchements dont les destinations sont temporairement indéfinies. Ces instructions sont conservées dans une liste d'instructions de branchement dont les champs étiquette seront remplis quand leurs valeurs seront déterminées.

La DDS pour un code **court-circuit** aura 2 attributs (hérités) : *vrai* qui donne l'étiquette de l'instruction à effectuer si l'expression est vraie, et *faux* l'étiquette si l'expression est fausse. On aura toujours les attributs (synthétisés) *code* et *place*.
Donc :

si L alors I_1 sinon I_2	$\rightarrow$		
			code de L
			avec branchement soit à $L.vrai$ *soit à* $L.faux$
		($L.vrai$)	*code de* I_1
			`aller a` $I.suivant$
		($L.faux$)	*code de* I_2
		($I.suivant$)	...

La DDS pour un code **court-circuit** est donné figure 9.3
Avec cette DDS, l'expression `si a<b ou c<d et e<f alors ...` sera traduite par

```
          si a<b aller a Lvrai
          aller a L1
(L1)      si c<d aller a L2
          aller a Lfaux
(l2)      si e<f aller a Lvrai
          aller a Lfaux
(Lvrai)   ...
          ...
          aller a Lsuiv
(Lfaux)   ...
          ...
(Lsuiv)   ...
```

On remarque que le code n'est pas optimal, la seconde instruction est inutile ici. Pour corriger cela, il faudra **optimiser** le code produit.

Production	Règle sémantique
$L \rightarrow Id\ oprel\ Id$	L.code= "si" $Id^{(1)}$.place *oprel*.valeur $Id^{(2)}$.place "aller a" L.vrai "aller a" L.faux
$I \rightarrow$ si L alors I sinon I	L.vrai = NouvEtiquette() L.faux = NouvEtiquette() $I^{(1)}$.suivant= $I^{(0)}$.suivant $I^{(2)}$.suivant= $I^{(0)}$.suivant $I^{(0)}.code$ = L.code "(" L.vrai ")" $I^{(1)}$.code "aller a" I.suivant "(" L.faux ")" $I^{(2)}$.code
$L \rightarrow L$ ou L	$L^{(1)}$.vrai = $L^{(0)}$.vrai $L^{(1)}$.faux = NouvEtiquette() $L^{(2)}$.vrai = $L^{(0)}$.vrai $L^{(2)}$.faux = $L^{(0)}$.faux $L^{(0)}$.code = $L^{(1)}$.code "(" $L^{(1)}$.faux ")" $L^{(2)}$.code
$L \rightarrow L$ et L	$L^{(1)}$.vrai = NouvEtiquette() $L^{(1)}$.faux = $L^{(0)}$.faux $L^{(2)}$.vrai = $L^{(0)}$.vrai $L^{(2)}$.faux = $L^{(0)}$.faux $L^{(0)}$.code = $L^{(1)}$.code "(" $L^{(1)}$.vrai ")" $L^{(2)}$.code
$L \rightarrow$ non L	$L^{(1)}$.vrai = $L^{(0)}$.faux $L^{(1)}$.faux = $L^{(0)}$.vrai $L^{(0)}$.code = $L^{(1)}$.code
$L \rightarrow$ vrai	L.code = "aller a" L.vrai
$L \rightarrow$ faux	L.code = "aller a" L.faux

FIG. 9.3 - *DDS de génération de code 3 adresses pour l'évaluation court circuit des expressions booléennes*

9.3 Optimisation de code

Le terme "optimisation" est abusif parce qu'il est rarement garanti que le code résultat soit le meilleur possible. On obtient une amélioration maximale pour un effort minimal si on peut identifier les parties du programme les plus fréquemment exécutées (on laisse alors tomber les autres parties). En effet, en général, dans la plupart des programmes, 90% du temps d'exécution est passé dans seulement 10% du code. Mais, of course, ce n'est pas évident de deviner quels sont les "points chauds" du programme[4].

9.3.1 Optimisations indépendantes de la machine cible

On effectue une *analyse des flux de données* pour chercher

- s'il existe un chemin d'exécution le long duquel la valeur d'une variable serait utilisée sans avoir été préalablement initialisée
- s'il existe du code "mort", c'est à dire une partie du programme qui n'est jamais atteinte
- s'il existe des variables ayant toujours une même valeur tout le long d'une partie du programme (*propagation des constantes*)
- les invariants de boucle, c'est à dire les expressions à l'intérieur d'une boucle qui ne dépendent d'aucune variable succeptible d'être modifiée dans la boucle, pour les déplacer devant la boucle
- si une expression est calculée plusieurs fois sans qu'entre temps les composantes de l'expression se trouvent modifiées (*élimination des expressions communes*)
- etc ...

Certaines de ces optimisations peuvent être effectuées avant la génération de code intermédiaire.
En outre, ces optimisations n'ont rien d'obligatoire, cela dépend du compilateur que l'on veut avoir. Il faut toujours se demander si l'amélioration de la qualité du code justifie ou non le temps passé sur le problème[5].

9.3.2 Optimisations dépendantes de la machine cible

- **allocation et assignation de registres**
- tenir compte des modes d'adressage
- remplacer des instructions générales par des instructions plus efficaces, spécifiques à la machine cible
- exploiter les possibilités de paralellisme du processeur cible
- etc. ...

9.3.3 Graphe de flot de contrôle

Pour réaliser les optimisations de code, il nous faut une représentation pratique du code intermédiaire. On utilisera le graphe de flot de contrôle.

Définition 9.3 *Un **bloc de base** est une séquence d'instructions consécutives contenant **au plus** une jonction au début (une arrivée de flot d'exécution) et un embranchement à la fin (un départ du flot d'exécution).*

Algorithme de partitionnement d'un programme en blocs de base :

1. on détermine les instructions de tête de chaque bloc de la manière suivante
   1.1. la première instruction est une instruction de tête
   1.2. toute instruction qui peut être atteinte par un branchement conditionnel ou inconditionnel est une instruction de tête
   1.3. toute instruction qui suit immédiatement un branchement conditionnel ou inconditionnel est une instruction de tête
2. chaque bloc de base débute avec son instruction de tête et se poursuit jusquà la première instruction qui précède une autre instruction de tête.

Exemple.

```
(1) i:=3                    (7)  si t2<b aller a (10)
(2) t1:=4*i                 (8)  t2:=t2+3
(3) t2:=a[t1]               (9)  aller a (7)
(4) j:=2                    (10) b:=b-j
(5) j:=j+1                  (11) aller a (5)
(6) si j>100 aller a (12)   (12) a[t1]:=t2
```

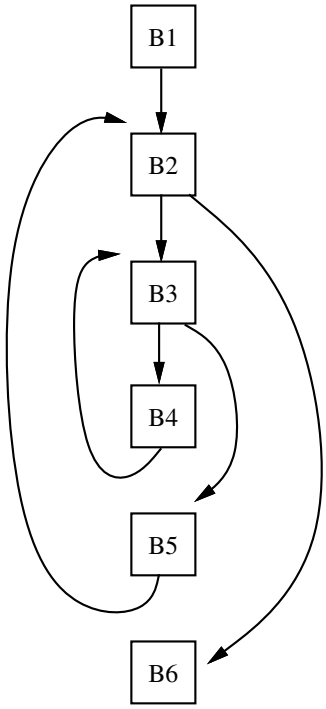
4. déja, on a intérêt à faire un effort pour les corps de boucle
5. signalons aussi que certaines optimisations souhaitées sont en fait des problèmes NP-complets ...

Les instructions de tête sont (1), (5), (7), (8) et (12). Les blocs sont $B_1 = \{(1),(2),(3),(4)\}$, $B_2 = \{(5),(6)\}$, $B_3 = \{(7)\}$, $B_4 = \{(8),(9)\}$, $B_5 = \{(10),(11)\}$ et $B_6 = \{(12)\}$.

Définition 9.4 *Le **graphe de flot de contrôle** est le graphe dont les noeuds sont les blocs de base et tel qu'il y a un arc du bloc B_i au bloc B_j si et seulement si l'une des deux conditions suivante est réalisée*

- *la dernière instruction de B_i est un branchement (conditionnel ou inconditionnel) à la première instruction de B_j*
- *B_j suit immédiatement B_i dans l'ordre du programme **et** B_i ne se termine pas par un branchement **in**conditionnel*

Sur l'exemple :



9.3.4 Elimination des sous-expressions communes

• Exemple 1 : soit le bloc de base

```
a:=b+c
b:=a-d
c:=b+c
d:=a-d
```

La deuxième et la dernière expression calculent la même chose, par conséquent on pourrait transformer ce bloc en

```
a:=b+c
b:=a-d
c:=b+c
d:=b
```

Attention, la première et la troisième expression ne calculent pas la même chose, car `b` est modifé entre temps !
• Exemple2 : soit le bloc de base

```
t3:=4*i
x:=a[t3]
t4:=4*i
a[t4]:=y
```

La première et la troisième expression sont identiques. Comme en plus c'est un calcul intermédiaire, on n'a même pas besoin de garder `t4`, il suffit de remplacer `t4` par `t3` à chacune de ses utilisations.

```
t3:=4*i
x:=a[t3]
a[t3]:=y
```

• Exemple3 : soit le bloc de base

```
t4:=4*i
t5:=a[t4]
t6:=4*i
t7:=a[t6]
a[t8]:=t7
```

On peut éliminer la troisième instruction :

```
t4:=4*i
t5:=a[t4]
t7:=a[t4]
a[t8]:=t7
```

On voit alors aparaître de nouveau une expression commune, que l'on peut encore éliminer :

```
t4:=4*i
t5:=a[t4]
a[t8]:=t5
```

• Exemple4 : Attention aux tableaux !!

```
...
x:=a[t1]
...
a[t2]:=...
...
y:=a[t1]
```

Ici, on ne peut pas considérer que l'expression `a[t1]` est commune, même si `t1` n'est pas modifié entre les deux expressions, car le tableau `a`, lui, est modifié et il n'est pas évident du tout (en général) de savoir si `t2` peut être égal à `t1` ou pas.

Bref, pour faire ça de manière générale, il nous faut parcourir le graphe de flot de contrôle et récolter les informations suivantes

- **les expressions produites** : `x op y` est produite par un bloc si elle est évaluée dans le bloc et si ni `x` ni `y` ne sont redéfinis ensuite dans ce bloc
- **les expressions supprimées** : `x op y` est supprimée par un bloc B si `x` ou `y` est redéfini dans le bloc sans que l'expression soit recalculé après (à l'intérieur du bloc)
- **les expressions disponibles** : `x op y` est disponible en un point p du programme si tous les chemins depuis le début du programme jusqu'à p l'évalue et si, aprs la dernière de ces évaluations sur chacun de ces chemins, il n'y a pas de nouvelle affectation de `x` ou `y`

• Algorithme de calcul de Prod(B) : expressions produites par le bloc B

parcourir le bloc instruction par instruction 1. au départ Prod(B)= $\emptyset$ 2. à chaque instruction du type `x:=y op z` 2.1. ajouter l'expression `y op z` à Prod(B) 2.2. enlever de Prod(B) toute expression contenant `x`

Exemple.

instruction	Prod
	$\emptyset$
`a:=b+c`	`b+c`
`e:=a+d`	`b+c a+d`
`b:=a-d`	`a+d a-d`
`c:=b+c`	`a+d a-d`
`d:=a-d`	$\emptyset$

• Algorithme de calcul de Supp(B) : expressions supprimées par le bloc B

Soit U l'ensemble de toutes les expressions du programme 1. au départ Supp(B)= $\emptyset$ 2. pour chaque instruction du bloc du type `x:=y op z`, mettre dans Supp(B) toutes les expressions de U qui contiennent **`x` à condition** qu'elle ne soit pas recalculée plus loin dans le bloc

Exemple.
On suppose que U= {`4*i j+3 j+4 2*j t[k] a-l t1-7 m+1 m-1 t[a] 2*a l*3`} et que l'on a le bloc B :

```
t1:=m-1
i:=m+1
j:=t[a]
k:=2*a
t2:=l*3
t3:=2*j
k:=t[k]
```

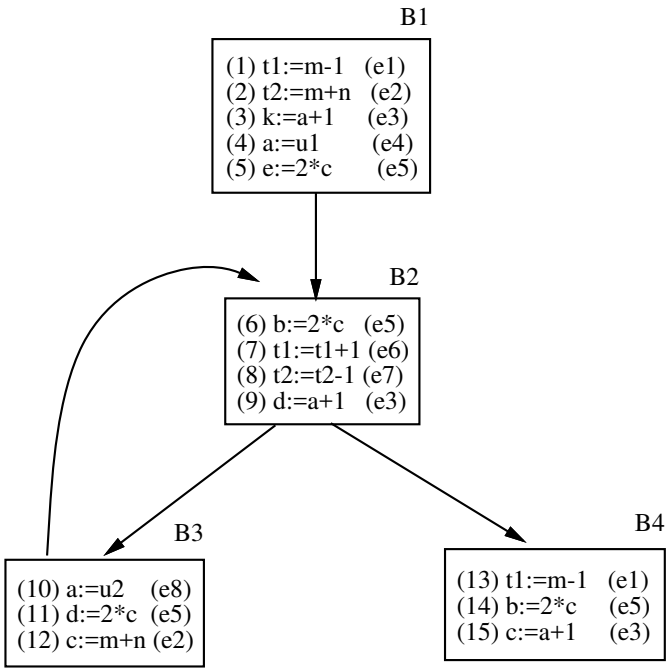
On obtient successivement :

Supp(B)= $\emptyset$
Supp(B)= {`t1-7`}
Supp(B)= {`t1-7 4*i`}
Supp(B)= {`t1-7 4*i j+3 j+4`}
Supp(B)= {`t1-7 4*i j+3 j+4`}
Supp(B)= {`t1-7 4*i j+3 j+4`}
Supp(B)= {`t1-7 4*i j+3 j+4`}
Supp(B)= {`t1-7 4*i j+3 j+4 t[k]`}

• Algorithme de calcul des In(B) et Ex(B) : expressions disponibles à l'entrée et à la sortie du bloc B

In(B_1) = $\emptyset$
Ex(B_1) = Prod(B_1)
pour chaque bloc B≠B_1 initialiser
  Ex(B)=U−Supp(B)
finpour
repeter
  pour chaque bloc B≠B_1 faire
    $In(B) = \bigcap_{P \text{ prédecesseur de } B} Ex(P)$
    Ex(B)=Prod(B)∪(In(B)−Supp(B))
  finpour
jusqu'a plus de changements

Exemple. (les numéros à la fin des instructions correspondent à une numérotation des expressions)



U= $\{e1, e2, e3, e4, e5, e6, e7, e7\}$
Calcul des Prod et des Supp :

bloc	Prod	Supp
B1	$e1$ $e2$ $e4$ $e5$	$e6$ $e7$ $e3$
B2	$e5$ $e3$	$e6$ $e7$
B3	$e8$ $e2$	$e3$ $e5$
B4	$e1$ $e3$	$e6$ $e5$

Calcul des In et Ex :

initialisations :

bloc	In	Ex
B1	$\emptyset$	$e1\ e2\ e4\ e5$
B2		$e1\ e2\ e3\ e4\ e5\ e8$
B3		$e1\ e2\ e4\ e6\ e7\ e8$
B4		$e1\ e2\ e3\ e4\ e7\ e8$

passe 1

- In(B2)=Ex(B1)$\cap$Ex(B3)= $\{e1, e2, e4\}$
  Ex(B2)=Prod(B2)$\cup$(In(B2)$-$Supp(B2))= $\{e3, e5\} \cup (\{e1, e2, e4\} - \{e6, e7\}) = \{e1, e2, e3, e4, e5\}$
- In(B3)=Ex(B2)= $\{e1, e2, e3, e4, e5\}$
  Ex(B3)=Prod(B3)$\cup$(In(B3)$-$Supp(B3))= $\{e2, e8\} \cup (\{e1, e2, e3, e4, e5\} - \{e3, e5\}) = \{e1, e2, e4, e8\}$
- In(B4)=Ex(B2)= $\{e1, e2, e3, e4, e5\}$
  Ex(B4)=Prod(B4)$\cup$(In(B4)$-$Supp(B4))= $\{e1, e3\} \cup (\{e1, e2, e3, e4, e5\} - \{e5, e6\}) = \{e1, e2, e3, e4\}$

passe 2

- In(B2)=Ex(B1)$\cap$Ex(B3)= $\{e1, e2, e4\}$ inchangé, donc Ex aussi
- In(B3)=Ex(B2)= $\{e1, e2, e3, e4, e5\}$ inchangé, donc Ex aussi
- In(B4)=Ex(B2)= $\{e1, e2, e3, e4, e5\}$ inchangé, donc Ex aussi

terminé, on a

bloc	In
B1	$\emptyset$
B2	$e1, e2, e4$
B3	$e1, e2, e3, e4, e5$
B4	$e1, e2, e3, e4, e5$

- Algorithme d'élimination des sous-expressions communes globales

> Pour chaque instruction `(i) x:=y op z` du bloc B telle que `y op z` est disponible **et** ni `y` ni `z` ne sont modifiées avant `(i)` dans B
>
> 1. chercher la(es) dernière(s) évaluation(s) de `y op z` en remontant le graphe de flot de contrôle et en visitant tous les chemins possibles
> 2. créer une nouvelle variable `u`
> 3. remplacer la(es) dernière(s) évaluation(s) `w:=y op z` par
>
> ```
> u:=y op z
> w:=u
> ```
>
> 4. remplacer l'instruction `(i)` par
>
> ```
> (i) x:=u
> ```

Sur l'exemple : les expressions à étudier sont $e1, e2, e3$ et $e5$

- $e1$: apparaît dans l'instruction (13) du bloc B4
  est-elle disponible? $e1 \in$In(B4), oui
  la dernière évaluation c'est l'instruction (1) qui devient donc
  `(1) t3:=m-1`
  `(1') i:=t3`
  et l'instruction (13) devient : `(13) i:=t3`
- $e2$: apparaît dans l'instruction (12) du bloc B3
  est-elle disponible? $e2 \in$In(B3), oui
  la dernière évaluation c'est l'instruction (2) qui devient donc
  `(2) t4:=m+n`
  `(2') j:=t4`
  et l'instruction (12) devient : `(12) c:=t4`
- $e3$: apparaît dans (9) de B2 et (15) de B4
  - (9) de B2
    est-elle disponible? $e3 \in$In(B2)?, non
    donc on ne peut pas l'éliminer
  - (15) de B4
    est-elle disponible? $e3 \in$In(B4), oui
    la dernière évaluation c'est l'instruction (9) qui devient donc
    `(9) t5:=a+1`
    `(9') d:=t5`
    et l'instruction (15) devient : `(15) c:=t5`
- $e5$: apparaît dans (6) de B2, (11) de B3 et (14) de B4
  - (6) de B2
    est-elle disponible? $e5 \in$In(B2)?, non

donc on ne peut pas l'éliminer

- (11) de B3
  est-elle disponible? $e5 \in$In(B3), oui
  la dernière évaluation c'est l'instruction (6) qui devient donc
  `(6) t6:=2*c`
  `(6') b:=t6`
  et l'instruction (11) devient : `(11) d:=t6`
- (14) de B4
  est-elle disponible? $e5 \in$In(B4), oui
  la dernière évaluation c'est l'instruction (6) qui devient donc (attention il y a déja une instruction `(6')`)
  `(6) t7:=2*c`
  `(6'') t6:=t7`
  et l'instruction (14) devient : `(14) b:=t7`

Bref, on obtient le graphe de la figure 9.4

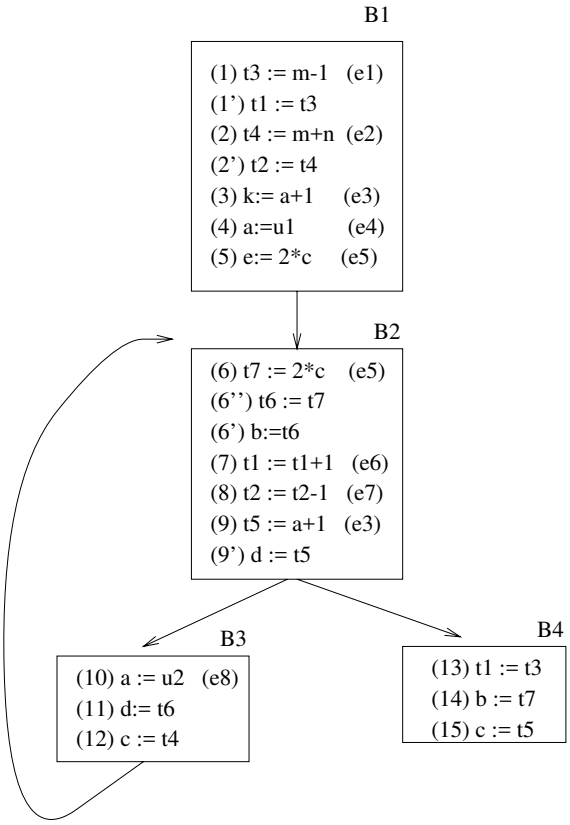


FIG. 9.4 - *après élimination des expressions communes*

On remarque que l'introduction de `t7` est totalement inutile, on aurait pu utiliser simplement `t6`. C'est l'algorithme de *propagation des copies* qui va se charger d'éliminer ce `t7`.

9.3.5 Propagation des copies

Les instructions de copie `(i) x:=y` peuvent être éliminées en remplaçant `x` par `y` dans chaque expression utilisant `x` à un point p si et seulement si

- l'instruction (i) est la seule définition de `x` qui atteint le point p
- sur chaque chemin de (i) à p (y compris les boucles) il n'y a aucune affectation de `y`

On laisse tomber mais c'est facile.

9.3.6 Calculs invariants dans les boucles

On cherche à sortir des boucles les calculs qui sont effectués à l'intérieur d'une boucle alors qu'ils n'évoluent jamais.
Exemple 1. Le graphe de la figure 9.5(a) peut se tranformer en le graphe 9.5(b)

Définition 9.5 *un point p du graphe de flot de contrôle* ***domine*** *un point q si tout chemin partant du point initial du graphe et arrivant à q passe par p*

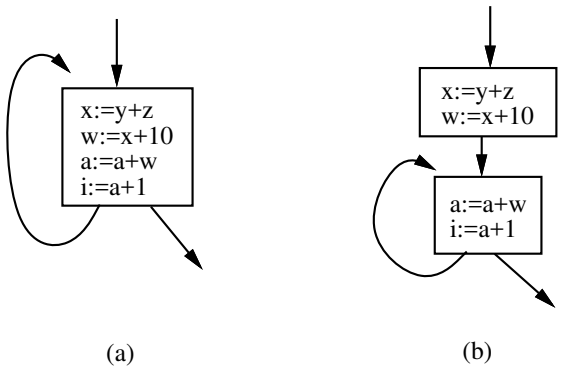


FIG. 9.5 - *Invariants de boucle (a) à l'origine (b) après déplacement de code*

Définition 9.6 *une* ***boucle*** *est une suite de blocs (ou un seul bloc) avec un en-tête qui domine tous les autres blocs de la boucle et telle qu'il existe au moins une manière de revenir vers l'en-tête à partir de n'importe quel bloc de la boucle*

Exemple 2 : dans la figure 9.6, les blocs B2-B3-B4 forment une boucle.

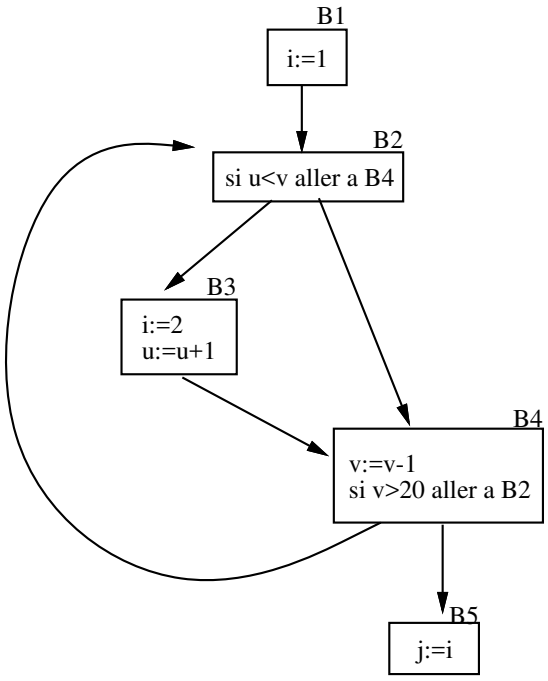


FIG. 9.6 - *déplacement de code impossible*

Mais on ne peut pas déplacer l'instruction `i:=2` de B3 hors de la boucle, car on ne passe pas forcemment par B3 quand on passe dans la boucle.

9.3.7 Interprétation abstraite

L'interprétation abstraite de programmes est une technique qui permet d'exprimer facilement les optimisations que nous avons vu jusqu'ici. Elle s'appuit sur des méthodes formelles de description de sémantique des langages : sémantique dénotationnelle ou sémantique opérationnelle.
C'est une méthode très puissante mais qui demande de connaître beaucoup de notions abstraites avant de la maîtriser. Nous n'en parlerons donc pas plus ici [6].

9.4 Génération de code

Traduction du code intermédiaire en code cible. Pose des problèmes spécifiques à la machine cible. Ce n'est pas notre problème.

6. pourtant, je vous jure, c'est vachement puissant

Index