

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Structure d'un compilateur</b>	<b>3</b>
<b>3</b>	<b>Analyse lexicale</b>	<b>4</b>
3.1	Unités lexicales et lexèmes . . . . .	4
3.2	Théorie des langages: expressions régulières . . . . .	4
3.3	Mise en œuvre d'un analyseur lexical . . . . .	4
<b>4</b>	<b>L'outil (f)lex</b>	<b>5</b>
4.1	Structure du fichier de spécifications (f)lex . . . . .	5
4.2	Les expressions régulières (f)lex . . . . .	6
4.3	Variables et fonctions prédéfinies . . . . .	6
4.4	Options de compilation flex . . . . .	7
<b>5</b>	<b>Analyse syntaxique</b>	<b>8</b>
5.1	Grammaires et Arbres de dérivation . . . . .	8
5.2	Analyse descendante . . . . .	8
5.2.1	Grammaire LL(1) . . . . .	9
5.3	Analyse ascendante . . . . .	10
<b>6</b>	<b>L'outil yacc/bison</b>	<b>11</b>
6.1	Structure du fichier de spécifications bison . . . . .	11
6.2	Attributs . . . . .	12
6.3	Communication avec l'analyseur lexical: yylval . . . . .	12
6.4	Variables, fonctions et actions prédéfinies . . . . .	13
6.5	Conflits shift-reduce et reduce-reduce . . . . .	13
6.5.1	Associativité et priorité des symboles terminaux . . . . .	13
6.6	Récupération des erreurs . . . . .	14
6.7	Options de compilations Bison . . . . .	14
6.8	Exemples de fichier .y . . . . .	14
<b>7</b>	<b>Théorie des langages: les automates</b>	<b>16</b>
7.1	Classification des grammaires . . . . .	16
7.2	Automate à états finis . . . . .	16
7.2.1	Automates finis déterministes (AFD) . . . . .	16
7.2.2	Minimisation d'un AFD . . . . .	17
7.3	Les automates à piles . . . . .	17
<b>8</b>	<b>Traduction dirigée par la syntaxe</b>	<b>18</b>
8.1	Définition dirigée par la syntaxe . . . . .	18
8.2	Evaluation des attributs . . . . .	18
<b>9</b>	<b>Génération de code</b>	<b>19</b>
9.1	Code à 3 adresses . . . . .	19
9.2	Optimisation de code à 3 adresses . . . . .	19

# Chapitre 1

## Introduction

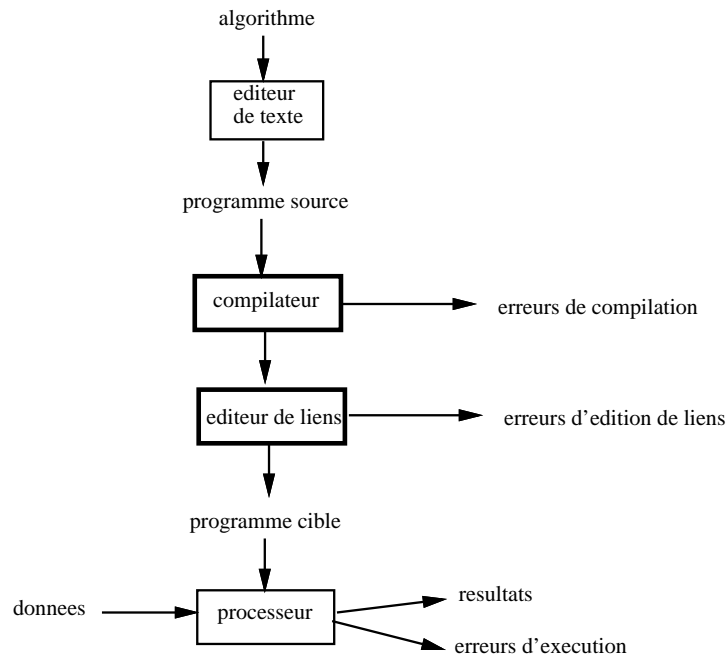


FIG. 1.1 - Chaîne de développement d'un programme

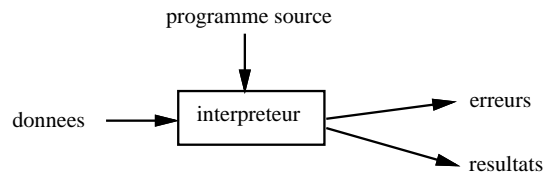


FIG. 1.2 - Interpréteur

- 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

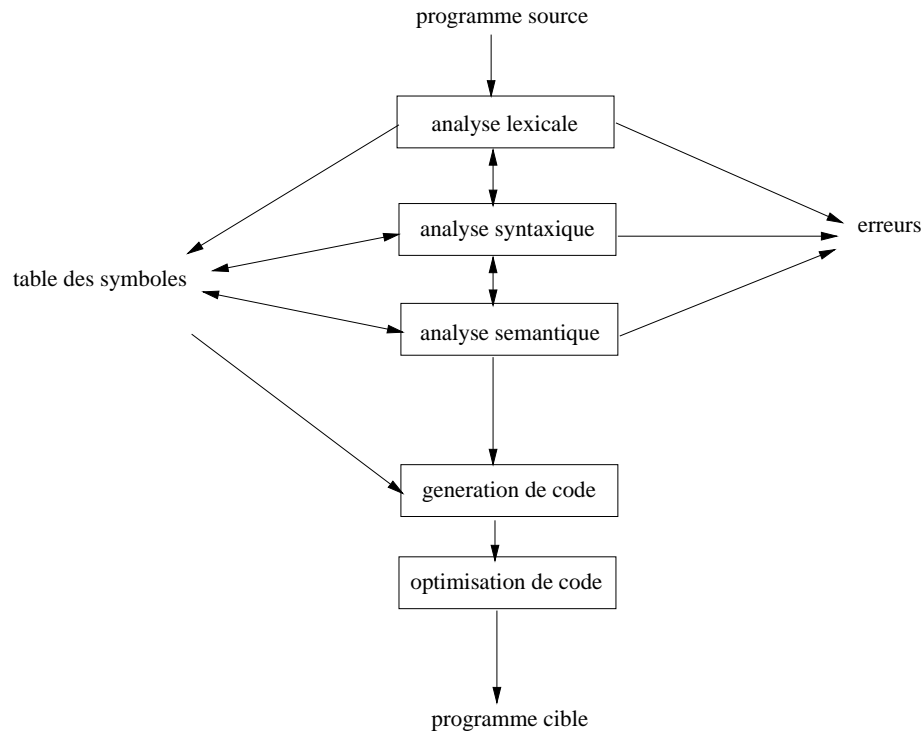


FIG. 2.1 - Structure d'un compilateur

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	

# Chapitre 3

## Analyse lexicale

### 3.1 Unités lexicales et lexèmes

- Une *unité lexicale* est une suite de caractères qui a une signification collective.
- 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.
- On appelle *lexème* toute suite de caractère qui concorde avec le modèle d'une unité lexicale.

### 3.2 Théorie des langages : expressions régulières

On appelle *alphabet* un ensemble fini non vide  $\Sigma$  de symboles (*lettres* de 1 ou plusieurs caractères).

On appelle *mot* toute séquence finie d'éléments de  $\Sigma$ . On note  $\varepsilon$  le *mot vide*.

On note  $\Sigma^*$  l'ensemble infini contenant tous les mots possibles sur  $\Sigma$ .

On note  $\Sigma^+$  l'ensemble des mots non vides que l'on peut former sur  $\Sigma$ , c'est à dire  $\Sigma^+ = \Sigma^* - \{\varepsilon\}$

On note  $|m|$  la *longueur* du mot  $m$ , c'est à dire le nombre de symboles de  $\Sigma$  composant le mot.

On note  $\Sigma^n$  l'ensemble des mots de  $\Sigma^*$  de longueur  $n$ .

On appelle *langage* sur un alphabet  $\Sigma$  tout sous-ensemble de  $\Sigma^*$ .

Opérations sur les langages :

**union :**  $L_1 \cup L_2 = \{w \text{ tq } w \in L_1 \text{ ou } w \in L_2\}$

**intersection :**  $L_1 \cap L_2 = \{w \text{ tq } w \in L_1 \text{ et } w \in L_2\}$

**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 :**  $L^* = \sum_{n \geq 0} L^n$

**Définition 3.1** Un langage régulier  $L$  sur un alphabet  $\Sigma$  est défini récursivement de la manière suivante :

- $\{\varepsilon\}$  est un langage régulier sur  $\Sigma$
- Si  $a$  est une lettre de  $\Sigma$ ,  $\{a\}$  est un langage régulier sur  $\Sigma$
- Si  $R$  est un langage régulier sur  $\Sigma$ , alors  $R^n$  et  $R^*$  sont des langages réguliers sur  $\Sigma$
- Si  $R_1$  et  $R_2$  sont des langages réguliers sur  $\Sigma$ , alors  $R_1 \cup R_2$  et  $R_1 R_2$  sont des langages réguliers
- il n'y a pas d'autres langages réguliers sur  $\Sigma$

**Définition 3.2** Les expressions régulières (E.R.) sur un alphabet  $\Sigma$  et les langages qu'elles décrivent sont définis récursivement de la manière suivante :

- $\varepsilon$  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  $R \cup S$
- 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

### 3.3 Mise en œuvre d'un analyseur lexical

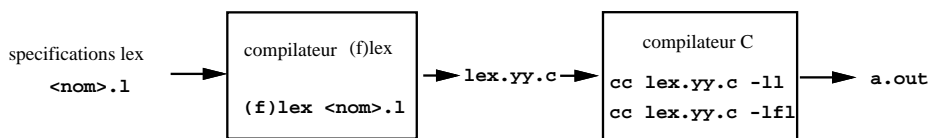
**Théorème 3.3** Un langage régulier est reconnu par un automate à états fini.

# Chapitre 4

## L'outil (f)lex

De nombreux outils ont été bâtis 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, reconnaît 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'exécutable obtenu lit le texte d'entrée caractère par caractère jusqu'à ce qu'il trouve le **plus long** préfixe du texte d'entrée qui correspond 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

```
exp1      action1  
exp2      action2  
⋮
```

Les `expi` sont des expressions régulières `(f)lex` et doivent commencer en colonne 0. Les `actioni` 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 contiennent les routines C définies par l'utilisateur et éventuellement une fonction `main()` si celle par défaut ne convient pas.

<code>c</code>	tout caractère <i>c</i> qui n'est pas un méta-caractère	<code>a</code>
<code>\c</code>	si <i>c</i> n'est pas une lettre minuscule, le caractère <i>c</i> littéralement	<code>\+</code>
<code>"s"</code>	la chaîne de caractère <i>s</i> littéralement	<code>"abc*+"</code>
<code>r<sub>1</sub>r<sub>2</sub></code>	<i>r<sub>1</sub></i> suivie de <i>r<sub>2</sub></i>	<code>ab</code>
<code>.</code>	n'importe quel caractère, excepté le caractère "à la ligne"	<code>a.b</code>
<code>^</code>	<b>comme premier caractère</b> de l'expression, signifie le début de ligne	<code>^abc</code>
<code>\$</code>	<b>comme dernier caractère</b> de l'expression, signifie la fin de la ligne	<code>abc\$</code>
<code>[s]</code>	n'importe lequel des caractères constituant la chaîne <i>s</i>	<code>[abc]</code>
<code>[^s]</code>	n'importe lequel des caractères à l'exception de ceux constituant la chaîne <i>s</i>	<code>[^abc]</code>
<code>r*</code>	0 ou plusieurs occurrences de <i>r</i>	<code>a*</code>
<code>r+</code>	1 ou plusieurs occurrences de <i>r</i>	<code>a+</code>
<code>r?</code>	0 ou 1 occurrence de <i>r</i>	<code>a?</code>
<code>r{m}</code>	<i>m</i> occurrences de <i>r</i>	<code>a{3}</code>
<code>r{m,n}</code>	<i>m</i> à <i>n</i> occurrences de <i>r</i>	<code>a{5,8}</code>
<code>r<sub>1</sub> r<sub>2</sub></code>	<i>r<sub>1</sub></i> ou <i>r<sub>2</sub></i>	<code>a b</code>
<code>r<sub>1</sub>/r<sub>2</sub></code>	<i>r<sub>1</sub></i> si elle est suivie de <i>r<sub>2</sub></i>	<code>ab/cd</code>
<code>(r)</code>	<i>r</i>	<code>(a b)?c</code>
<code>\n</code>	caractère "à la ligne"	
<code>\t</code>	tabulation	
<code>{}</code>	pour faire référence à une définition régulière	<code>{nombre}</code>
<code>«EOF»</code>	fin de fichier (UNIQUEMENT avec <code>flex</code> )	<code>«EOF»</code>

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)|(machin*)`

`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 reconnaît pas "truc" avec lex, mais le reconnaît 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.

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 reconnaît 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'explicitier !
```

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

### 5.1 Grammaires et Arbres de dérivation

**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  $V_T \cap V_N = \emptyset$
- $S$  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  $\alpha$  ( $\alpha \in (V_T \cup V_N)^+$ ) peut être **remplacée** par la séquence de symboles  $\beta$  ( $\beta \in (V_T \cup V_N)^*$ ).

$\alpha$  est appelée **partie gauche** de la production, et  $\beta$  **partie droite** de la production. On parle de **dérivation** de  $\alpha$  en  $\beta$

**Définition 5.3** Le langage généré par  $G$  est  $L(G) = \{w \in (V_T)^* \mid S \xrightarrow{*} w\}$

**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  $\varepsilon$
- les noeuds sont des symboles non-terminaux
- les fils d'un noeud  $\alpha$  sont  $\beta_0, \dots, \beta_n$  si et seulement si  $\alpha \rightarrow \beta_0 \dots \beta_n$  est une production

### 5.2 Analyse descendante

#### Calcul de PREMIER

$\text{PREMIER}(\alpha)$  est l'ensemble de tous les **terminaux** qui peuvent **commencer** une chaîne qui se dérive de  $\alpha$

$\text{PREMIER}(\alpha) = \{a \mid \exists \beta \alpha \xrightarrow{*} a\beta\}$

1. Si  $X$  est un terminal,  $\text{PREMIER}(X) = \{X\}$ .
  2. Si  $X$  est un non-terminal et  $X \rightarrow Y_1 Y_2 \dots Y_n$  est une production de la grammaire (avec  $Y_i$  symbole terminal ou non-terminal) alors
    - ajouter les éléments de  $\text{PREMIER}(Y_1)$  **sauf**  $\varepsilon$  dans  $\text{PREMIER}(X)$
    - s'il existe  $j$  ( $j \in \{2, \dots, n\}$ ) tel que pour tout  $i = 1, \dots, j-1$  on a  $\varepsilon \in \text{PREMIER}(Y_i)$ , alors ajouter les éléments de  $\text{PREMIER}(Y_j)$  **sauf**  $\varepsilon$  dans  $\text{PREMIER}(X)$
    - si pour tout  $i = 1, \dots, n$   $\varepsilon \in \text{PREMIER}(Y_i)$ , alors ajouter  $\varepsilon$  dans  $\text{PREMIER}(X)$
  3. Si  $X$  est un non-terminal et  $X \rightarrow \varepsilon$  est une production, ajouter  $\varepsilon$  dans  $\text{PREMIER}(X)$
- Recommencer jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles  $\text{PREMIER}$ .

#### Calcul de SUIVANT

$\text{SUIVANT}(A)$  est l'ensemble de tous les symboles terminaux  $a$  qui peuvent apparaître immédiatement à droite de  $A$  dans une dérivation:  $S \xrightarrow{*} \alpha A a \beta$

1. Ajouter un marqueur de fin de chaîne (symbole \$ par exemple) à  $\text{SUIVANT}(S)$  (où  $S$  est l'axiome de départ de la grammaire)
  2. Pour chaque production  $A \rightarrow \alpha B \beta$  où  $B$  est un non-terminal, alors ajouter le contenu de  $\text{PREMIER}(\beta)$  à  $\text{SUIVANT}(B)$ , **sauf**  $\varepsilon$
  3. Pour chaque production  $A \rightarrow \alpha B$ , alors ajouter  $\text{SUIVANT}(A)$  à  $\text{SUIVANT}(B)$
  4. Pour chaque production  $A \rightarrow \alpha B \beta$  avec  $\varepsilon \in \text{PREMIER}(\beta)$ , ajouter  $\text{SUIVANT}(A)$  à  $\text{SUIVANT}(B)$
- Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles  $\text{SUIVANT}$ .

## Construction de la table d'analyse

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

### Exemple

La figure 5.1 donne la table d'analyse de la grammaire ETF des expressions arithmétiques :

$$\left\{ \begin{array}{ll} E \rightarrow TE' & T' \rightarrow *FT' | \varepsilon \\ E' \rightarrow +TE' | \varepsilon & F \rightarrow (E) | \text{nb} \\ T \rightarrow FT' & \end{array} \right.$$

	nb	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
$F$	$F \rightarrow \text{nb}$			$F \rightarrow (E)$		

FIG. 5.1 - Table LL de la grammaire ETF

### 5.2.1 Grammaire LL(1)

**Définition 5.5** On appelle **grammaire LL(1)** une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.

**Théorème 5.6** Une grammaire **ambigüe** ou **récursive à gauche** ou **non factorisée à gauche** n'est pas LL(1)

**Définition 5.7** Une grammaire est dite **ambigüe** s'il existe un mot de  $L(G)$  ayant plusieurs arbres syntaxiques.

**Définition 5.8** Une grammaire est **immédiatement récursive à gauche** si elle contient un non-terminal  $A$  tel qu'il existe une production  $A \rightarrow A\alpha$ . Une grammaire est **récursive à gauche** si elle contient un non-terminal  $A$  tel qu'il existe une dérivation  $A \xrightarrow{+} A\alpha$

#### Élimination de la récursivité à gauche immédiate :

Remplacer toute règle de la forme  $A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_p$  par les deux règles :

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_p A'$$

$$A' \rightarrow \varepsilon | \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A'$$

#### Élimination de la récursivité à gauche :

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

#### Factorisation à gauche

Pour chaque non-terminal  $A$   
 trouver le plus long préfixe  $\alpha$  commun à deux de ses productions (ou plus)  
 Si  $\alpha \neq \varepsilon$ , remplacer  $A \rightarrow \alpha\beta_1 | \dots | \alpha\beta_n | \gamma_1 | \dots | \gamma_p$  (où les  $\gamma_i$  ne commencent pas par  $\alpha$ ) par  
    $A \rightarrow \alpha A' | \gamma_1 | \dots | \gamma_p$   
    $A' \rightarrow \beta_1 | \dots | \beta_n$   
 finsi  
 finpour  
 Recommencer jusqu'à ne plus en trouver.

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			

FIG. 5.2 - Table d'analyse SLR des expressions arithmétiques

### 5.3 Analyse ascendante

#### 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  $A \rightarrow \alpha.B\beta$   
pour chaque production  $B \rightarrow \gamma$   
rajouter (s'il n'y est pas déjà) l'item  $B \rightarrow .\gamma$  dans Fermeture( $I$ )  
finpour
- 3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

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

$$\Delta(I, X) = \text{Fermeture}(\text{tous les items } A \rightarrow \alpha X \beta) \text{ où } A \rightarrow \alpha.X\beta \in I$$

#### Collection des items d'une grammaire $G$ :

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

#### Construction de la table d'analyse SLR:

- 1- Construire la collection d'items  $\{I_0, \dots, I_n\}$
- 2- l'état  $i$  est construit 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  $A \rightarrow \alpha$ . (sauf  $A = S'$ ) contenu dans  $I_i$ :  
mettre **reduire**  $A \rightarrow \alpha$  dans **chaque** case  $M[i, a]$  où  $a \in \text{SUIVANT}(A)$
  - d) si  $S' \rightarrow S \in I_i$ : mettre **accepter** dans la case  $M[i, \$]$

Exemple: grammaire des expressions arithmétiques

- $$\left\{ \begin{array}{lll} (1) & E \rightarrow E + T & (3) & T \rightarrow T * F & (5) & F \rightarrow (E) \\ (2) & E \rightarrow T & (4) & T \rightarrow F & (6) & F \rightarrow \text{nb} \end{array} \right.$$

La table d'analyse LR de cette grammaire est donnée figure 5.2

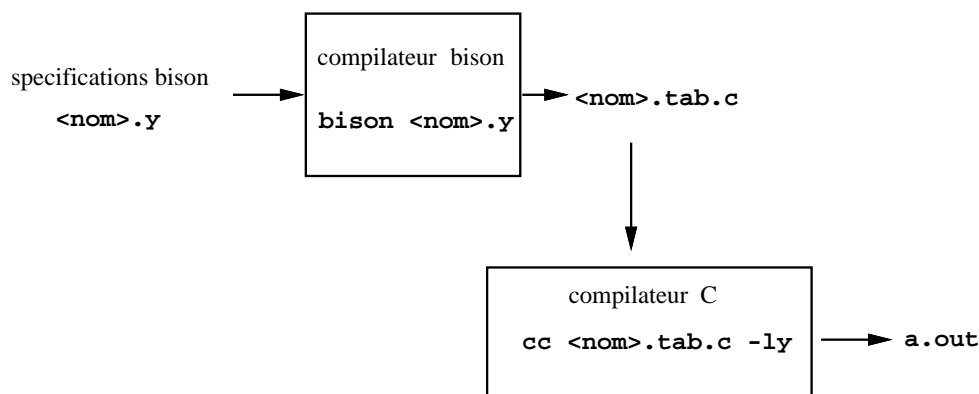
# Chapitre 6

## 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, reconnaît 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

```

| 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 `nom.l`. Dans ce cas, il suffira d'inclure le fichier `lex.yy.c` produit par `(f)lex` et de rajouter la bibliothèque `(f)lex` lors de la compilation C du fichier `nom.tab.c` (avec `cc nom.tab.c -ly -l(f)l`).

## 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 <chaîne> 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 <chaîne> expr
```

## 6.4 Variables, fonctions et actions prédéfinies

`YYACCEPT`: instruction qui permet de stopper l'analyseur syntaxique. Dans ce cas, `yyparse` retourne la valeur 0 (succès).

`YYABORT`: instruction qui permet également de stopper l'analyseur. `yyparse` retourne alors 1, ce qui peut être utilisé pour signifier l'échec de l'analyseur.

`yyparse()`: appel de l'analyseur syntaxique.

`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 **décaler** 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 cela, il faut compiler avec l'option `-v`. Le fichier contenant la table s'appelle `y.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 une réduction  $A \rightarrow \alpha$  et un décalage d'un symbole d'entrée  $a$ , est alors résolu en appliquant les règles suivantes :

- si la priorité de la production  $A \rightarrow \alpha$  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.

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  $A \rightarrow \alpha_1|\alpha_2|\dots$  une production

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

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  $\beta$ .

Exemple: La production

$$instr \rightarrow \mathbf{error};$$

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 `yyerror` replace l'analyseur dans le mode normal de fonctionnement c'est à dire que l'analyse syntaxique n'est pas interrompue.

## 6.7 Options de compilations Bison

- l'option `-v` produit un fichier `nom.output` contenant un descriptif des conflits shift/reduce et reduce/reduce détectés et indique de quelle façon ils ont été résolus. Il est **fortement conseillé** de consulter ce fichier afin de vérifier que les conflits sont résolus comme on le désire.
- l'option `-d` produit un fichier `nom.tab.h` contenant les définitions (sous forme de `#define`) des unités lexicales rencontrées et du type `YYSTYPE` s'il est redéfini.

## 6.8 Exemples de fichier .y

Cet exemple reconnaît 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`,

fichier doit se terminer par \$.

Cet exemple utilise la fonction `yylex` générée par le compilateur `flex` à partir du fichier de spécification `exemple2.1` suivant

```
%{
#include <stdlib.h>
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 `yacc` 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;}
         ;

%%
#include "lex.yy.c"
```

Le Makefile pour compiler tout ça est

```
CC=gcc
exemple2 : lex.yy.c exemple2.tab.c
          $(CC) exemple2.tab.c -o exemple2 -ly -lfl
lex.yy.c : exemple2.1
          flex exemple2.1
exemple2.tab.c : exemple2.y lex.yy.c
             bison exemple2.y
```

# Chapitre 7

## Théorie des langages : les automates

### 7.1 Classification des grammaires

- Grammaire de **type 3 (régulière)**: Les productions sont de la forme  $A \rightarrow wB$  ou bien  $A \rightarrow w$
- Grammaire de **type 2 (hors contexte)**: Les productions sont de la forme  $A \rightarrow \beta$
- Grammaire de **type 1 (contextuelle)**: Les productions sont de la forme  $\alpha \rightarrow \beta$  avec  $|\alpha| \leq |\beta|$ . Les productions  $S \rightarrow \varepsilon$  sont autorisées si  $S \in V_N$  n'apparaît pas dans la partie droite d'une production
- Grammaire de **type 0**: Les productions sont de la forme  $\alpha \rightarrow \beta$

**Théorème 7.1** *Les automates à états finis sont des reconnaissseurs pour les langages réguliers.*

**Théorème 7.2** *Les automates à pile sont des reconnaissseurs pour les langages hors contexte.*

### 7.2 Automate à états finis

**Définition 7.3** *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  $\Sigma$  des symboles d'entrée
- une fonction de transition  $\Delta$  qui à tout couple formé d'un état  $e$  et d'un symbole  $a$  de  $\Sigma$  fait correspondre un ensemble (éventuellement vide) d'états:  $\Delta(e, a) = \{e_1, \dots, e_n\}$

**Définition 7.4** *Le langage reconnu par un automate est l'ensemble des chaînes qui permettent de passer de l'état initial à un état terminal.*

#### 7.2.1 Automates finis déterministes (AFD)

**Définition 7.5** *On appelle  $\varepsilon$ -transition, une transition par le symbole  $\varepsilon$  entre deux états.*

**Définition 7.6** *Un automate fini est dit déterministe lorsqu'il ne possède pas de  $\varepsilon$ -transition et lorsque pour chaque état  $e$  et pour chaque symbole  $a$ , il y a au plus un arc étiqueté  $a$  qui quitte  $e$*

**Déterminisation d'un AFN.**

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

1. Partir de l' $\varepsilon$ -fermeture de l'état initial
2. Rajouter dans la table de transition toutes les  $\varepsilon$ -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. Renommer alors les états.

**Définition 7.7** *On appelle  $\varepsilon$ -fermeture de l'ensemble d'états  $T = \{e_1, \dots, e_n\}$  l'ensemble des états accessibles depuis un état  $e_i$  de  $T$  par des  $\varepsilon$ -transitions*

Calcul de l' $\varepsilon$ -fermeture de  $T = \{e_1, \dots, e_n\}$

```

Initialiser  $\varepsilon$ -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  $\varepsilon$ -transition entre  $p$  et  $e$  faire
    Si  $e$  n'est pas déjà dans  $\varepsilon$ -fermeture( $T$ )
      ajouter  $e$  à  $\varepsilon$ -fermeture( $T$ )
      empiler  $e$  dans  $P$ 
    finsi
  finpour
fin tantque

```

### 7.2.2 Minimisation d'un AFD

Principe : 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 pour 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

## 7.3 Les automates à piles

**Définition 7.8** *Un automate à pile est défini par :*

- un ensemble fini d'état  $E$
- un état initial  $e_o \in E$
- un ensemble fini d'états finaux  $T$
- un alphabet  $\Sigma$  des symboles d'entrée
- un alphabet  $\Gamma$  des **symboles de pile**
- un symbole  $\tau \in \Gamma$  de **fond de pile**
- une relation de transition  $\Delta : E \times \Sigma^* \times \Gamma^* \rightarrow E \times \Gamma^*$   
 $\Delta(p, a, \alpha) = (q, \beta)$ , signifie que si on est dans l'état  $p$  avec le mot  $\alpha$  en sommet de pile et la lettre  $a$  en entrée, on peut passer dans l'état  $q$  en remplaçant  $\alpha$  par  $\beta$  dans la pile.

**Définition 7.9** *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.*

# Chapitre 8

## Traduction dirigée par la syntaxe

### 8.1 Définition dirigée par la syntaxe

- Chaque symbole de la grammaire (terminal ou non) possède un ensemble d'**attributs**.
- Chaque règle de production de la grammaire possède un ensemble de **règles sémantiques** (suites d'instructions algorithmiques) qui permettent de calculer la valeur des attributs associés aux symboles apparaissant dans la production.

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

On notera  $X.a$  l'attribut  $a$  du symbole  $X$ . S'il y a plusieurs symboles  $X$  dans une production, on les notera  $X^{(1)}, \dots, X^{(n)}$ , et  $X^{(0)}$  s'il est en partie gauche.

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

### 8.2 Evaluation des attributs

**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. // Un attribut est dit *hérité* lorsqu'il est calculé pour un non terminal en partie droite 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 en un noeud d'un attribut synthétisé se calcule en fonction des attributs de ses **fil**s (calcul des feuilles vers la racine). Celle d'un attribut hérité se calcule en fonction des attributs des **frères et du père** (calcul de la racine vers les feuilles).

**Définition 8.4** 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  $a$ .

**Propriété 8.5** Si le graphe de dépendances contient un cycle, alors le calcul des attributs est impossible.

# Chapitre 9

## Environnement d'exécution

### 9.1 Organisation de la mémoire

(modèle général)

On divise le bloc de mémoire allouées à l'exécution par le système d'exploitation cible en 4 zones : cf figure ??

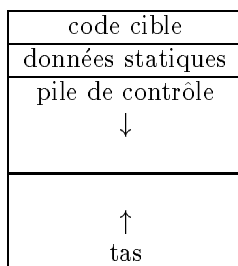
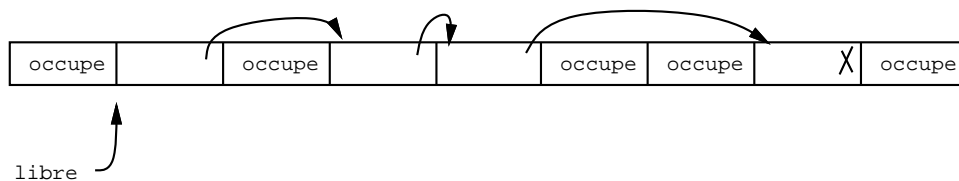


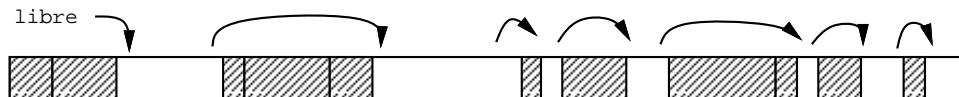
FIG. 9.1 - *Modèle général d'organisation de la mémoire*

### 9.2 Allocation dynamique : gestion du tas

- 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, ...)



- Libération implicite  
Modèle de format de bloc :

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

- Compression : rassembler toute la mémoire libre en un seul grand bloc.

# Chapitre 10

## Génération de code

### 10.1 Code à 3 adresses

- Séquences d'instructions **numérotées**, parmi

affectation binaire	(num) <b>x := y op z</b>
affectation unaire	(num) <b>x := opu z</b>
affectation indicée	(num) <b>y[i] := z</b>
copie	(num) <b>x:=y</b>
branchement inconditionnel	(num1) <b>aller a (num2)</b>
branchement conditionnel	(num1) <b>si x oprel y aller a (num2)</b>
lecture	(num) <b>lire x</b>
écriture	(num) <b>crire y</b>

- avec
- op** opérateur binaire comme +, \*, -, /, ET, OU, ≥, [...]...
  - opu** opérateur unaire comme -, √, NON, ...
  - oprel** opérateur relationnel (=, <, ≥, ≠, ...)
  - x** adresse<sup>1</sup> de variable ou registre
  - y, z** adresse de variable, registre ou constante

- le nombre de registres disponibles est **illimité**

La figure 9.1 donne une TDS générant le code à 3 adresses des expressions arithmétiques.

La figure 9.2 donne une TDS générant le code à 3 adresses de l'évaluation paresseuse des expressions booléennes.

La figure 9.3 donne une TDS générant le code à 3 adresses de l'évaluation paresseuse des expressions booléennes.

### 10.2 Optimisation de code à 3 adresses

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

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

FIG. 10.1 - TDS de génération de code 3 adresses pour les expressions arithmétiques

Production	Règle sémantique												
$I \rightarrow \text{si } L \text{ alors } I \text{ sinon } I$	$I^{(1)}.suivant = I^{(0)}.suivant$ $I^{(2)}.suivant = I^{(0)}.suivant$ $Vrai = \text{NouvEtiquette}()$ $Faux = \text{NouvEtiquette}()$ $I^{(0)}.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td></td> <td><math>L.code</math></td> </tr> <tr> <td></td> <td>"si" <math>L.place</math> "vrai aller a" <math>Vrai</math></td> </tr> <tr> <td></td> <td>"aller a" <math>Faux</math></td> </tr> <tr> <td>"(" <math>Vrai</math> ")"</td> <td><math>I^{(1)}.code</math></td> </tr> <tr> <td></td> <td>"aller a" <math>I^{(1)}.suivant</math></td> </tr> <tr> <td>"(" <math>Faux</math> ")"</td> <td><math>I^{(2)}.code</math></td> </tr> </table>		$L.code$		"si" $L.place$ "vrai aller a" $Vrai$		"aller a" $Faux$	"(" $Vrai$ ")"	$I^{(1)}.code$		"aller a" $I^{(1)}.suivant$	"(" $Faux$ ")"	$I^{(2)}.code$
	$L.code$												
	"si" $L.place$ "vrai aller a" $Vrai$												
	"aller a" $Faux$												
"(" $Vrai$ ")"	$I^{(1)}.code$												
	"aller a" $I^{(1)}.suivant$												
"(" $Faux$ ")"	$I^{(2)}.code$												
$L \rightarrow L \text{ ou } L$	$L^{(0)}.place = \text{NouvRegistre}()$ $L^{(0)}.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td><math>L^{(1)}.code</math></td> </tr> <tr> <td><math>L^{(2)}.code</math></td> </tr> </table> $L^{(0)}.place := " L^{(1)}.place \text{ ou } L^{(2)}.place$	$L^{(1)}.code$	$L^{(2)}.code$										
$L^{(1)}.code$													
$L^{(2)}.code$													
$L \rightarrow Id \text{ oprel } Id$	$L.place = \text{NouvRegistre}()$ $L.code = L.place := " Id.place \text{ oprel.valeur } Id.place$												
$L \rightarrow \text{non } L$	$L^{(0)}.place = \text{NouvRegistre}()$ $L^{(0)}.code = L^{(0)}.place := \text{non" } L^{(1)}.place$												
$\vdots$	etc.												

FIG. 10.2 - TDS de génération de code 3 adresses pour l'évaluation paresseuse des expressions booléennes

Production	Règle sémantique							
$L \rightarrow Id \text{ oprel } Id$	$L.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td>"si" <math>Id^{(1)}.place \text{ oprel.valeur } Id^{(2)}.place</math> "aller a" <math>L.vrai</math></td> </tr> <tr> <td>"aller a" <math>L.faux</math></td> </tr> </table>	"si" $Id^{(1)}.place \text{ oprel.valeur } Id^{(2)}.place$ "aller a" $L.vrai$	"aller a" $L.faux$					
"si" $Id^{(1)}.place \text{ oprel.valeur } Id^{(2)}.place$ "aller a" $L.vrai$								
"aller a" $L.faux$								
$I \rightarrow \text{si } L \text{ alors } I \text{ sinon } I$	$L.vrai = \text{NouvEtiquette}()$ $L.faux = \text{NouvEtiquette}()$ $I^{(1)}.suivant = I^{(0)}.suivant$ $I^{(2)}.suivant = I^{(0)}.suivant$ $I^{(0)}.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td><math>L.code</math></td> </tr> <tr> <td>"(" <math>L.vrai</math> ")"</td> <td><math>I^{(1)}.code</math></td> </tr> <tr> <td></td> <td>"aller a" <math>I.suivant</math></td> </tr> <tr> <td>"(" <math>L.faux</math> ")"</td> <td><math>I^{(2)}.code</math></td> </tr> </table>	$L.code$	"(" $L.vrai$ ")"	$I^{(1)}.code$		"aller a" $I.suivant$	"(" $L.faux$ ")"	$I^{(2)}.code$
$L.code$								
"(" $L.vrai$ ")"	$I^{(1)}.code$							
	"aller a" $I.suivant$							
"(" $L.faux$ ")"	$I^{(2)}.code$							
$L \rightarrow L \text{ ou } L$	$L^{(1)}.vrai = L^{(0)}.vrai$ $L^{(1)}.faux = \text{NouvEtiquette}()$ $L^{(2)}.vrai = L^{(0)}.vrai$ $L^{(2)}.faux = L^{(0)}.faux$ $L^{(0)}.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td><math>L^{(1)}.code</math></td> </tr> <tr> <td>"(" <math>L^{(1)}.faux</math> ")"</td> <td><math>L^{(2)}.code</math></td> </tr> </table>	$L^{(1)}.code$	"(" $L^{(1)}.faux$ ")"	$L^{(2)}.code$				
$L^{(1)}.code$								
"(" $L^{(1)}.faux$ ")"	$L^{(2)}.code$							
$L \rightarrow L \text{ et } L$	$L^{(1)}.vrai = \text{NouvEtiquette}()$ $L^{(1)}.faux = L^{(0)}.faux$ $L^{(2)}.vrai = L^{(0)}.vrai$ $L^{(2)}.faux = L^{(0)}.faux$ $L^{(0)}.code =$ <table style="display: inline-table; vertical-align: middle;"> <tr> <td><math>L^{(1)}.code</math></td> </tr> <tr> <td>"(" <math>L^{(1)}.vrai</math> ")"</td> <td><math>L^{(2)}.code</math></td> </tr> </table>	$L^{(1)}.code$	"(" $L^{(1)}.vrai$ ")"	$L^{(2)}.code$				
$L^{(1)}.code$								
"(" $L^{(1)}.vrai$ ")"	$L^{(2)}.code$							
$L \rightarrow \text{non } L$	$L^{(1)}.vrai = L^{(0)}.faux$ $L^{(1)}.faux = L^{(0)}.vrai$ $L^{(0)}.code = L^{(1)}.code$							
$L \rightarrow \text{vrai}$	$L.code = \text{"aller a" } L.vrai$							
$L \rightarrow \text{faux}$	$L.code = \text{"aller a" } L.faux$							

FIG. 10.3 - TDS de génération de code 3 adresses pour l'évaluation court circuit des expressions booléennes

- 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 jusqu'à la première instruction qui précède une autre instruction de tête.

**Définition 10.1** 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 **non conditionnel**
- Algorithme de calcul de  $\text{Prod}(B)$  : expressions **produites** par le bloc  $B$

parcourir le bloc instruction par instruction

1. au départ  $\text{Prod}(B) = \emptyset$
2. à chaque instruction du type  $x := y \text{ op } z$ 
  - 2.1. ajouter l'expression  $y \text{ op } z$  à  $\text{Prod}(B)$
  - 2.2. enlever de  $\text{Prod}(B)$  toute expression contenant  $x$

- Algorithme de calcul de  $\text{Supp}(B)$  : expressions **supprimées** par le bloc  $B$

Soit  $U$  l'ensemble de toutes les expressions du programme

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

- Algorithme de calcul des  $\text{In}(B)$  et  $\text{Ex}(B)$  : expressions **disponibles** à l'entrée et à la sortie du bloc  $B$

```

In(B1) = ∅
Ex(B1) = Prod(B1)
pour chaque bloc B ≠ B1 initialiser
    Ex(B) = U - Supp(B)
finpour
repete
    pour chaque bloc B ≠ B1 faire
        In(B) = ⋂P prédecesseur de B Ex(P)
        Ex(B) = Prod(B) ∪ (In(B) - Supp(B))
    finpour
jusqu'à plus de changements

```

- Algorithme d'élimination des sous-expressions communes globales

Pour chaque instruction (i)  $x := y \text{ op } z$  du bloc  $B$  telle que  $y \text{ op } z$  est disponible et ni  $y$  ni  $z$  ne sont modifiées avant (i) dans  $B$

1. chercher la(es) dernière(s) évaluation(s) de  $y \text{ op } z$  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)  $w := y \text{ op } z$  par
 

```

                u := y op z
                w := u
            
```
4. remplacer l'instruction (i) par
 

```

                (i) x := u
            
```

finpour

Exemple : l'élimination des sous-expressions communes globales du fragment de programme donné figure 9.4(a) donne le fragment de programme 9.4(b)

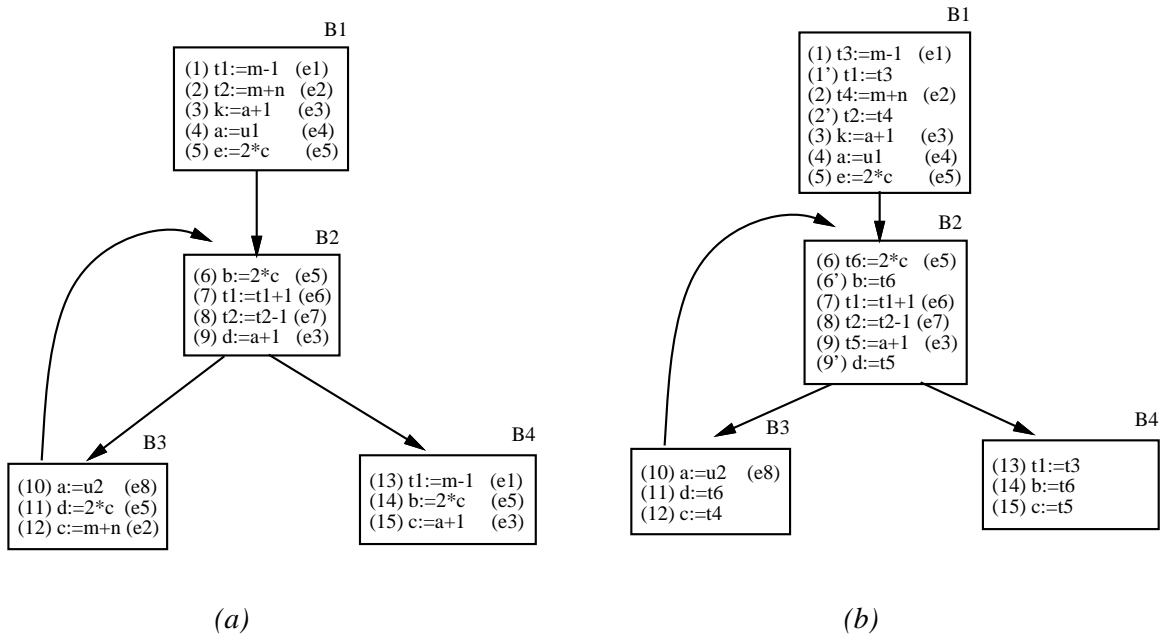


FIG. 10.4 - élimination des expressions communes : (a) avant (b) après