

# Exercices



HABIB ABDULRAB (INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE ROUEN)

CLAUDE MOULIN (UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE)

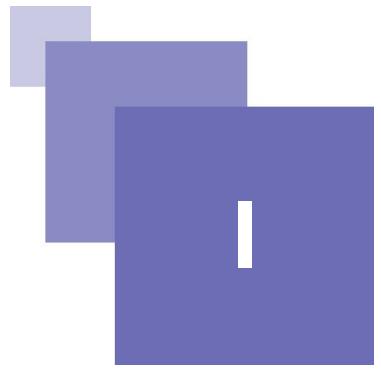
SID TOUATI (UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN EN YVELINES)

# Table des matières

<b>I - TD 0 - Introduction à la programmation assembleur</b>	<b>7</b>
A.Pré-requis du TD 0.....	7
B.Exercice 1.....	7
C.Exercice 2.....	9
D.Exercice 3.....	10
E.Exercice 4.....	10
F.Exercice 5.....	10
<b>II - TD1 - Analyse lexicale</b>	<b>13</b>
A.Exercice 1.....	13
B.Exercice 2.....	14
C.Exercice 3.....	14
D.Exercice 4.....	15
E.Exercice 5.....	15
<b>III - TD 2 - Analyse syntaxique</b>	<b>17</b>
A.Exercice 1 : Analyse descendante.....	17
B.Exercice 2 : Grammaire.....	17
C.Exercice 3 : IF ... THEN ... ELSE (LL).....	18
D.Exercice 4 : IF ... THEN ... ELSE (LR).....	19
E.Exercice 5 : SLR, LALR.....	19
<b>IV - TD3 - Analyse sémantique</b>	<b>21</b>
A.Exercice 1 : Grammaire LR d'expressions arithmétiques.....	21
B.Exercice 2 : Grammaire LL d'expressions arithmétiques.....	21
<b>V - TD 4 - Actions sémantiques et Yacc (table Aes symboles simple)</b>	<b>23</b>

TD 0 - Introduction à la programmation assembleur	
A.Pré-requis du TD4.....	<b>23</b>
B.Exercice 1.....	<b>23</b>
C.Exercice 2 : Utilisation de Yacc.....	<b>24</b>
D.Exercice 3 : Utilisation de Yacc (suite).....	<b>24</b>
<b>VI - TD 5 - Actions sémantiques et Yacc (Gestion des types)</b>	<b>27</b>
A.Pré-requis du TD 5 .....	<b>27</b>
B.Exercice 1.....	<b>27</b>
C.Exercice 2.....	<b>28</b>
<b>VII - TD 6 - Tables de symboles et types.</b>	<b>29</b>
A.Pré-requis du TD 6 .....	<b>29</b>
B.Exercice 1.....	<b>29</b>
C.Exercice 2.....	<b>30</b>
<b>VIII - TD 7 - Génération de code</b>	<b>31</b>
A.Pré-requis du TD 7.....	<b>31</b>
B.Exercice 1.....	<b>31</b>
C.Exercice 2.....	<b>32</b>
<b>IX - TD 8 - Génération de code</b>	<b>35</b>
A.Pré-requis du TD 8 .....	<b>35</b>
B.Exercice 1.....	<b>35</b>
C.Exercice 2.....	<b>36</b>
D.Exercice 3.....	<b>36</b>
<b>X - TD 9 - Génération de code et optimisation.</b>	<b>39</b>
A.Pré-requis du TD 9.....	<b>39</b>
B.Exercice 1.....	<b>39</b>
C.Exercice 2.....	<b>40</b>
<b>Solution des exercices rédactionnels</b>	<b>43</b>

# TD 0 - Introduction à la programmation assembleur



Pré-requis du TD 0	7
Exercice 1	7
Exercice 2	9
Exercice 3	10
Exercice 4	10
Exercice 5	10

## A. Pré-requis du TD 0

- Avoir suivi le cours d'introduction à la compilation qui explique les rudiments de la programmation assembleur (x86).
- Pour l'architecture i386, avoir étudié les différences entre la syntaxe asm AT&T et la syntaxe intel. Voir le document donné en annexe (cf. Annexe TD0).
- Pour faire l'exercice 1, il faudrait étudier le document donné en annexe décrivant la sémantique de chaque instruction asm i386. Attention, il y a une différence entre la syntaxe intel et la syntaxe AT&T (adoptée par gnu gcc).

## B. Exercice 1

Traduire le programme assembleur ci-dessous en langage C.

Question

[Solution n°1 p 37]

```
chaine :  
    .string "%d\n"  
.globl main  
    .type main, @function  
main :  
    pushl %ebp  
    movl %esp, %ebp  
    subl $24, %esp  
    movl $0, tab  
    movl $1, -4(%ebp)  
loop1 :  
    cmpl $99, -4(%ebp)  
    jle body1  
    jmp skip1  
body1 :  
    movl -4(%ebp), %edx  
    leal 0(%edx, 4), %eax  
    movl $tab, %edx  
    movl $1, (%eax, %edx)  
    incl -4(%ebp)  
    jmp loop1  
skip1 :  
    movl $2, -4(%ebp)  
loop2 :  
    cmpl $9, -4(%ebp)  
    jle body2  
    jmp skip2  
body2 :  
    movl -4(%ebp), %eax  
    movl %eax, %edx  
    leal (%edx, %eax), %ecx  
    movl %ecx, -8(%ebp)  
loop3 :  
    cmpl $99, -8(%ebp)  
    jle body3  
    jmp skip3
```

```

body3 :
    movl -8(%ebp), %edx
    leal 0(%edx, 4), %eax
    movl $tab, %edx
    movl $0, (%eax, %edx)
    movl -4(%ebp), %eax
    addl %eax, -8(%ebp)
    jmp loop3
skip3 :
    incl -4(%ebp)
    jmp loop2
skip2 :
    movl $0, -4(%ebp)
loop4 :
    cmpl $99, -4(%ebp)
    jle body4
    jmp skip4
body4 :
    movl -4(%ebp), %edx
    leal 0(%edx, 4), %eax
    movl $tab, %edx
    cmpl $1, (%eax, %edx)
    jne else
    addl $-8, %esp
    movl -4(%ebp), %eax
    pushl %eax

    pushl $chaine
    call printf
    addl $16, %esp
else :
    incl -4(%ebp)
    jmp loop4
skip4 :
    leave
    ret
.comm tab, 400, 32

```

## C.Exercice 2

Générer le code assembleur du programme C ci-dessous.

Question

[Solution n°2 p 39]

```
main() {
    int i, j;
    j = i;
    i = 5;
}
```

## D.Exercice 3

Générer le code assembleur du programme C ci-dessous.

Question

[Solution n°3 p 40]

```
main() {
    int i, s;
    s = 0;
    for ( i = 0; i < 100; i + + )
        s + = i;
}
```

## E.Exercice 4

Générer le code assembleur du code C ci-dessous. On supposera que le tableau est rangé e, lignes.

Question

[Solution n°4 p 40]

```
main() {
    int i, tab1[10][10], tab2[10][10];
    for ( i = 1; i < 10; i + + )
        tab1[i][i] = tab2[i][i];
}
```

## F.Exercice 5

Dans la déclaration suivante, déterminez la formule donnant l'adresse de la case mémoire référencée par l'expression: tab[i][j][k].

Question

[Solution n°5 p 41]

```
struct t {
    int a, b;
};
struct t tab[n1][n2][n3];
```



# TD1 - Analyse lexicale

Exercice 1	13
Exercice 2	14
Exercice 3	14
Exercice 4	15
Exercice 5	15

## A.Exercice 1

Trouver un automate reconnaissant :

Question 1

[Solution n°6 p 42]

Le langage  $A^*/L$ ; où L est le langage des mots ayant une seule occurrence de b

Question 2

[Solution n°7 p 42]

L'ensemble de nombres entiers (signés ou non) sur l'alphabet  $A = \{+, -, 0, 1, \dots, 9\}$ .

Question 3

[Solution n°8 p 42]

L'ensemble de nombres rationnels (signés ou non) sur l'alphabet  $A = \{+, -, /, 0, 1, \dots, 9\}$ .

Les langages suivants sont-ils reconnaissables ?

Question 4

[Solution n°9 p 42]

$L =$  l'ensemble des mots sur  $A = \{a, b\}$  terminant par aba.

i.e.  $L = \{waba / w \in A^*\}$ .

Question 5

[Solution n°10 p 43]

$L =$  l'ensemble des mots sur  $A = \{0, 1\}$  qui sont des représentations binaires des entiers naturels multiples de 3.

Un automate X est complet si pour chaque état  $q$  de X, et chaque lettre  $a$  de A, il existe une flèche de la forme  $(q, a, q')$ , où  $q'$  est un état de l'automate.

Question 6

[Solution n°11 p 43]

Montrer que pour chaque automate  $X$  on peut associer un automate  $X'$  noté  $Complet(X)$  t.q.:  
 $|X| = |Complet(X)|$

## B.Exercice 2

Les langages suivants sont-ils réguliers ?

### Question 1

[\[Solution n°12 p 44\]](#)

Le langage  $S = A^*/L$ ; où  $L$  est le langage des mots ayant une seule occurrence de  $b$

### Question 2

[\[Solution n°13 p 44\]](#)

L'ensemble  $L$  de nombres entiers (signés ou non) sur l'alphabet  $A = \{+, -, 0, 1, \dots, 9\}$

### Question 3

[\[Solution n°14 p 44\]](#)

L'ensemble  $L$  de nombres rationnels (signés ou non) sur l'alphabet  $A = \{+, -, /, 0, 1, \dots, 9\}$ .

### Question 4

[\[Solution n°15 p 44\]](#)

$L$  = l'ensemble  $L$  des mots sur  $A = \{a, b\}$  terminant par  $aba$ .

i.e.  $L = \{waba / w \in A^*\}$ .

### Question 5

[\[Solution n°16 p 44\]](#)

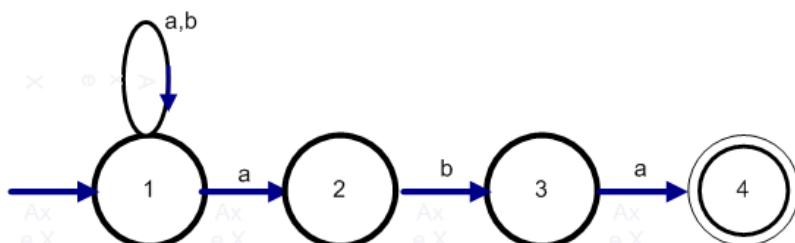
1.10)  $a^+b^* \cap \{a, b\}^*ba^*$

## C.Exercice 3

Déterminiser les automates suivants

### Question 1

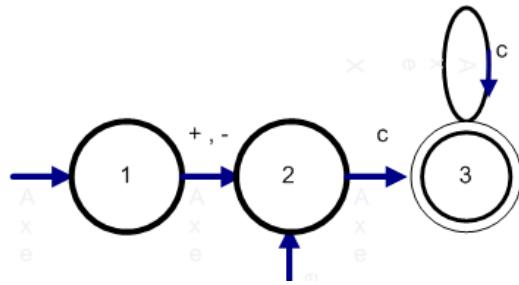
[\[Solution n°17 p 44\]](#)



### Question 2

[\[Solution n°18 p 44\]](#)





## D.Exercice 4

Si  $L_1$  et  $L_2$  sont deux langages réguliers:

### Question 1

[Solution n°19 p 45]

Calculer l'automate  $\text{intersection}(X_1, X_2)$  qui reconnaît  $L_1 \cap L_2$ . En déduire l'automate reconnaissant:  $a^+b^* \cap \{a, b\}^*ba^*$ .

### Question 2

[Solution n°20 p 46]

Calculer l'automate  $\text{complementaire}(X_1)$  qui reconnaît  $A^* \setminus L_1$ . En déduire l'automate reconnaissant:  $A^* \setminus a^*ba^*$ .

### Question 3

[Solution n°21 p 46]

En déduire que  $L_1 \cap L_2$  et  $A^* \setminus L_1$  sont réguliers.

## E.Exercice 5

### Question

[Solution n°22 p 46]

Trouver un scanner non déterministe qui associe à chaque mot  $m$  de  $\{a\}^+$ , de longueur  $n$ , le mot  $b^n$  si  $n$  est pair, le mot  $c^n$  sinon.

# TD 2 - Analyse syntaxique

III

Exercice 1 : Analyse descendante	17
Exercice 2 : Grammaire	17
Exercice 3 : IF ... THEN ... ELSE (LL)	18
Exercice 4 : IF ... THEN ... ELSE (LR)	19
Exercice 5 : SLR, LALR	19

## A.Exercice 1 : Analyse descendante

Soit la grammaire :

$$\begin{aligned} S &\mapsto AaB \\ A &\mapsto CB \mid Bb \mid \epsilon \\ B &\mapsto b \\ C &\mapsto c \mid \epsilon \end{aligned}$$

### Question 1

[Solution n°23 p 47]

Construire les ensembles PREMIER et SUIVANT pour cette grammaire.

### Question 2

[Solution n°24 p 47]

Établir la table d'analyse et montrer que cette grammaire n'est pas LL(1)

## B.Exercice 2 : Grammaire

Soit la grammaire d'expressions arithmétiques définie par les productions suivantes :

$$\begin{aligned} S &\mapsto E \ ; \ S \mid \epsilon \\ E &\mapsto E \ + \ T \mid E \ - \ T \mid T \\ T &\mapsto T \ * \ F \mid T \ / \ F \mid F \\ F &\mapsto \text{const} \mid ( \ E \ ) \end{aligned}$$

### Question 1

[Solution n°25 p 47]

Les terminaux de la grammaire sont : { ; + - \* / const () }

Donner les dérivations les plus à gauche pour les chaînes  $5 + 3 * 2;$  et  $3; 2/3 * (1 - 3);$

#### Question 2

[Solution n°26 p 48]

Cette grammaire est-t-elle récursive à gauche ? Si oui, transformez-la en grammaire récursive à droite.

#### Question 3

[Solution n°27 p 48]

On cherche maintenant à écrire un analyseur syntaxique descendant pour cette grammaire. Quelle grammaire utiliser ?

#### Question 4

[Solution n°28 p 48]

Simuler l'analyse de l'expression  $3 * (1 + 2)/(1);$  par un analyseur de type descendant.

## C.Exercice 3 : IF ... THEN ... ELSE (LL)

Soit la grammaire :

$$\begin{array}{lcl} 1 \quad <inst> & \mapsto & IF <expression> THEN <inst><else-inst> \\ 2 \quad <inst> & \mapsto & ID := ID \\ 3 \quad <else-inst> & \mapsto & ELSE <inst> \\ 4 \quad <else-inst> & \mapsto & \epsilon \\ 5 \quad <expression> & \mapsto & ID \end{array}$$

Les terminaux sont : *IF, THEN, ID, :=, ELSE*

L'axiome est  $<inst>$

#### Question 1

[Solution n°29 p 50]

Cette grammaire est-elle LL(1) ?

#### Question 2

[Solution n°30 p 50]

Comment lève-t-on généralement l'ambiguïté rencontrée ?

#### Question 3

[Solution n°31 p 51]

Analyser l'instruction suivante en utilisant le choix de la question précédente :

*IF x THEN IF y THEN i := j ELSE i := k*

## D.Exercice 4 : IF ... THEN ... ELSE (LR)

Soit la grammaire :

$$\begin{array}{lcl} <inst> & \mapsto & IF <expression> THEN <inst><else-inst> \\ <inst> & \mapsto & ID := ID \\ <else-inst> & \mapsto & ELSE <inst> \\ <else-inst> & \mapsto & \epsilon \\ <expression> & \mapsto & ID \end{array}$$



Les terminaux sont : *IF*, *THEN*, *ID*, *:=*, *ELSE*

Question 1

[*Solution n°32 p 51*]

Cette grammaire est-elle LR(0) ?

Question 2

[*Solution n°33 p 51*]

Cette grammaire est-elle SLR(1) ?

Question 3

[*Solution n°34 p 52*]

S'il existe des conflits quelle convention utiliser pour les supprimer ?

## E.Exercice 5 : SLR, LALR

Considérons la grammaire G suivante :

$$\begin{array}{l} S \xrightarrow{} Aa \\ S \xrightarrow{} bAc \\ S \xrightarrow{} dc \\ S \xrightarrow{} bda \\ A \xrightarrow{} d \end{array}$$

Question

[*Solution n°35 p 52*]

Montrer que cette grammaire est LALR(1) mais pas SLR(1).

# TD3 - Analyse sémantique

IV

Exercice 1 : Grammaire LR d'expressions arithmétiques.

21

Exercice 2 : Grammaire LL d'expressions arithmétiques.

21

## A.Exercice 1 : Grammaire LR d'expressions arithmétiques.

Soit la grammaire LR d'expressions arithmétiques définie par les productions numérotées comme suit :

- (1)  $E \xrightarrow{} E + T$
- (2)  $E \xrightarrow{} T$
- (3)  $T \xrightarrow{} T * F$
- (4)  $T \xrightarrow{} F$
- (5)  $F \xrightarrow{} ( E )$
- (6)  $F \xrightarrow{} \text{nb}$

### Question 1

[*Solution n°36 p 53*]

Programmez cette grammaire avec lex & yacc.

### Question 2

[*Solution n°37 p 54*]

Ajouter des actions sémantiques à la grammaire yacc pour calculer la valeur d'une expression arithmétique écrite avec la grammaire précédente.

## B.Exercice 2 : Grammaire LL d'expressions arithmétiques.

Soit la grammaire LL d'expressions arithmétiques définie par les productions numérotées comme suit :

- (1)  $E \rightarrow T R$
- (2)  $R \rightarrow + T R$
- (3)  $R \rightarrow$
- (4)  $T \rightarrow F P$
- (5)  $P \rightarrow * F P$
- (6)  $P \rightarrow$
- (7)  $F \rightarrow ( E )$
- (8)  $F \rightarrow \text{nb}$

### Question 1

[Solution n°38 p 55]

Proposer une grammaire AntLR pour représenter cette grammaire.

### Question 2

[Solution n°39 p 55]

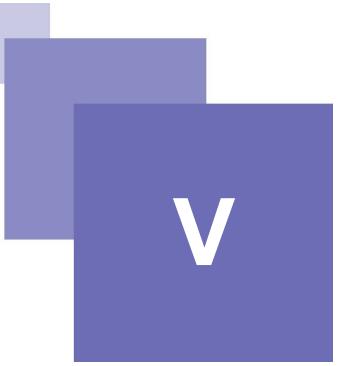
Ajouter des actions sémantiques en Java à la grammaire AntLR pour calculer la valeur d'une expression arithmétique écrite avec la grammaire précédente.

### Question 3

[Solution n°40 p 56]

Donner un programme Java permettant de lire un fichier contenant une expression arithmétique et utilisant le lexer et le parser créés à partir de la grammaire AntLR.

# TD 4 - Actions sémantiques et Yacc (table Aes symboles simple)



Pré-requis du TD4	23
Exercice 1	23
Exercice 2 : Utilisation de Yacc	24
Exercice 3 : Utilisation de Yacc (suite)	24

## A. Pré-requis du TD4

- Avoir suivi le cours de compilation sur la traduction dirigée par la syntaxe.
- Maîtriser les outils { lex & yacc}, sinon relire le manuel et refaire les TD précédents.

## B. Exercice 1

Afin de construire une calculatrice gérant les affectations de variables, on utilise la grammaire suivante :

$$\begin{array}{lcl} S & \mapsto & I ; S \mid \epsilon \\ I & \mapsto & \text{ident} := E \mid E \mid \epsilon \\ E & \mapsto & E + T \mid E - T \mid T \\ T & \mapsto & T * F \mid T / F \mid F \\ F & \mapsto & \text{ident} \mid \text{const} \mid ( E ) \end{array}$$

### Question 1

[Solution n°41 p 56]

Donner une interface simple de programmation d'une table de symboles adaptée à notre calculatrice.

### Question 2

[Solution n°42 p 57]

Dans le cas d'une traduction dirigée par la syntaxe S-attribuée, déterminer les

actions sémantiques associées à chaque production.

#### Question 3

Soit le texte  $a := 5 ; a * a + 2 ;$ . Dessiner l'arbre de dérivation correspondant en mentionnant les valeurs des attributs pour chaque symbole syntaxique (terminal et non terminal).

## C.Exercice 2 : Utilisation de Yacc

On souhaite toujours réaliser une calculatrice, mais en partant cette fois de la grammaire suivante:

$$E \mapsto E + E \mid E - E \mid E * E \mid E / E \mid - E \mid ( E ) \mid \text{const}$$

#### Question 1

[Solution n°43 p 57]

Cette grammaire est-elle ambiguë ? Si oui, donner un exemple d'entrée créant un conflit *shift/reduce*.

#### Question 2

[Solution n°44 p 57]

Ces conflits peuvent être résolus en exploitant l'associativité et les priorités usuelles des opérateurs \*, /, - et +.

L'opérateur unaire - possède la plus haute priorité.

Écrire un analyseur lexical (avec lex) et un analyseur syntaxique (avec yacc) pour cette calculatrice après avoir lever l'ambiguïté de la grammaire.

## D.Exercice 3 : Utilisation de Yacc (suite)

On ajoute maintenant à notre calculatrice la gestion de variables dont le nom est une lettre (en minuscule) de l'alphabet: il n'y a donc que 26 noms de variables possibles.

La calculatrice doit également accepter les nombres en virgule flottante et plusieurs calculs à la fois. Pour cela, on enrichit la grammaire:

$$\begin{aligned} L &\mapsto I \mid L ; I \\ I &\mapsto \text{ident} := E \mid E \\ E &\mapsto E + E \mid E - E \mid E * E \mid E / E \\ E &\mapsto - E \mid ( E ) \mid \text{const} \mid \text{ident} \end{aligned}$$

#### Question

[Solution n°45 p 59]

Modifier en conséquence les spécifications lex et yacc de la question précédente. On utilisera un tableau de typedouble[] pour stocker les valeurs des variables.



# TD 5 - Actions sémantiques et Yacc (Gestion des types)

Pré-requis du TD 5

27

Exercice 1

27

Exercice 2

28

## A. Pré-requis du TD 5

- Avoir compris le cours de compilation sur la traduction dirigée par la syntaxe.
- Avoir suivi le cours de compilation sur la table des symboles.

## B. Exercice 1

On reprend la grammaire de calculatrice suivante:

$$\begin{aligned} L &\mapsto I \mid L ; I \\ I &\mapsto \text{ident} := E \mid E \\ E &\mapsto E + E \mid E - E \mid E * E \mid E / E \\ E &\mapsto -E \mid (E) \mid \text{const} \mid \text{ident} \end{aligned}$$

On désire étendre le nom des variables à une suite quelconque de lettres de l'alphabet.

### Question 1

[Solution n°46 p 60]

A-t-on besoin d'une table de symboles ? Si oui, proposez plusieurs types d'implémentations possibles. Quelle structure de données est la plus utilisée ?

### Question 2

[Solution n°47 p 61]

Implémentez en langage C une table de symboles possédant l'interface suivante:

```
typedef struct symbole {
    char * nom;
    double valeur;
} symbole;
symbole * inserer( char * nom );
```

### Question 3

[Solution n°48 p 62]

Écrire des spécifications lex et yacc utilisant la table de symboles implémentée dans la question précédente.

## C.Exercice 2

Le but de cet exercice est d'analyser des déclarations de fonctions du type  
`int foo( int a, string b );`

On utilise pour cela la grammaire suivante:

$$\begin{array}{lcl} S & \mapsto & D \mid S D \\ D & \mapsto & T \text{ ident}(L) ; \mid T \text{ ident}() ; \\ L & \mapsto & P \mid L , P \\ P & \mapsto & T \text{ ident} \\ T & \mapsto & \text{int} \mid \text{string} \end{array}$$

### Question 1

[Solution n°49 p 66]

A l'aide de lex et yacc, écrivez un parseur analysant ces déclarations et affichant pour chaque déclaration un résumé du type:

*Fonction : foo*

*Type : int*

*Arguments : a (int), b (string)*

### Question 2

[Solution n°50 p 69]

Dans l'éventualité de déclarations multiples d'une fonction, on désire vérifier la cohérence des déclarations, et afficher au besoin des messages d'erreur. Modifiez votre parseur en conséquence.



# TD 6 - Tables de symboles et types.

Pré-requis du TD 6

29

Exercice 1

29

Exercice 2

30

## A. Pré-requis du TD 6

- Avoir compris le cours de compilation sur la traduction dirigée par la syntaxe.
- Avoir suivi le cours de compilation sur la table des symboles.

## B. Exercice 1

### Déclarations de variables

On souhaite écrire un compilateur pour un langage comportant des déclarations de variables locales à une procédure, du type:

```
int toto;
string titi;
string[10] tata;
...
```

On décrit la syntaxe de ces déclarations à l'aide de la grammaire suivante:

$$\begin{array}{lcl} L & \mapsto & D \mid LD \\ D & \mapsto & T Q \text{ ident} ; \\ T & \mapsto & \text{int} \mid \text{string} \\ Q & \mapsto & [\text{const}] \mid \epsilon \end{array}$$

On souhaite enregistrer chaque variable déclarée dans une table de symboles, et associer à chaque variable la position (relative) en mémoire dans laquelle elle sera stockée.

### Question 1

*[Solution n°51 p 75]*

Proposez une interface adaptée pour la table des symboles.

## Question 2

[Solution n°52 p 75]

Proposez des actions sémantiques (en syntaxe yacc) pour chaque production de la grammaire.

# C.Exercice 2

## Déclarations de variables (suite)

On reprend la grammaire de l'exercice précédent.

On souhaite maintenant traiter également des déclarations internes à un bloc (et dont la portée se limite à ce bloc), avec des blocs éventuellement imbriqués, comme dans l'exemple suivant:

```
{ int a;  
{ int c;  
int a; /* re-déclaration licite de 'a' */  
...  
}  
...  
}
```

On étend la grammaire de l'exercice précédent en ajoutant la règle:

$I \rightarrow \{LI\}|\{I\}|S$

où  $S$  désigne une *instruction* que l'on ne détaillera pas.

## Question 1

[Solution n°53 p 76]

Peut-on encore utiliser une table de symboles unique ?

Proposez un mécanisme permettant de gérer les déclarations internes à chaque bloc et écrivez les actions sémantiques de la grammaire.

## Question 2

[Solution n°54 p 77]

On désire afficher un message d'avertissement lorsqu'une variable déclarée dans un bloc en masque une autre déclarée dans un bloc englobant. Modifiez les actions sémantiques en conséquence.



# TD 7 - Génération de code

VIII

Pré-requis du TD 7

31

Exercice 1

31

Exercice 2

32

## A. Pré-requis du TD 7

- Assembleur i386 (revoir le TD1).
- Avoir compris les cours de compilation sur la traduction dirigée par la syntaxe.
- Avoir suivi le cours de compilation sur les codes intermédiaires et sur la génération de code.

## B. Exercice 1

### Un compilateur pour calculatrice

On reprend la grammaire des expressions arithmétiques vue précédemment :

$$\begin{aligned} S &\mapsto I ; S \mid \epsilon \\ I &\mapsto \text{ident} := E \mid E \mid \epsilon \\ E &\mapsto E + T \mid E - T \mid T \\ T &\mapsto T * F \mid T / F \mid F \\ F &\mapsto \text{ident} \mid \text{const} \mid ( E ) \end{aligned}$$

Au lieu de calculer les expressions lors de l'analyse syntaxique, on désire cette fois générer du code assembleur pour architecture i386.

Ce code devra calculer chaque expression et en afficher la valeur en appelant la fonction `printf` de la librairie C.

#### Question 1

[Solution n°55 p 78]

Proposez une structure de données simple permettant de représenter et manipuler un programme assembleur.

#### Question 2

[Solution n°56 p 78]

Notre code assembleur doit pouvoir lire et affecter des variables.

Proposez plusieurs mécanismes d'allocation des variables et de génération de code correspondante.

### Question 3

[Solution n°57 p 78]

On décide dans un premier temps de générer un code exploitant essentiellement la pile d'exécution du processeur i386, en adoptant les conventions suivantes:

- Les opérandes d'une expression sont toujours présentes sur la pile (l'opérande la plus à droite sur le sommet de la pile).
- Le résultat d'une expression est systématiquement empilé.

Écrire une spécification yacc correspondante. A cet effet, nous supposons que chaque symbole syntaxique non terminal a un attributCode qui contient le code généré.

### Question 4

[Solution n°58 p 82]

On modifie légèrement nos conventions de génération de code en exploitant davantage les registres du processeur:

- Les opérandes d'une expression sont placées dans les registres %eax et %ecx.
- Le résultat d'une expression est systématiquement placé dans le registre %eax.

Modifiez en conséquence votre spécification yacc.

### Question 5

[Solution n°59 p 85]

Le code assembleur ainsi généré est-il directement exécutable ?

Expliquez comment obtenir un binaire exécutable, et complétez éventuellement votre spécification yacc.

### Question 6

[Solution n°60 p 86]

Si la fonction `printf` utilisait le passage par *valeur* au lieu du passage par *adresse* des chaînes de caractères, comment votre programme serait-il modifié ? on suppose dans ce cas que les chaînes de caractères ont une taille fixe de 128 octets.

## C.Exercice 2

### Appels de fonctions

On souhaite maintenant utiliser dans la calculatrice de l'exercice précédent des fonctions externes (comme par exemple `abs`, `rand` de la librairie C). Ces fonctions devront avoir un prototype C de la forme:

```
int foo( int p1, int p2, ..., int pk )
```

On étend la grammaire de la calculatrice:

$$F \rightarrow \text{ident} \mid \text{const} \mid (E) \mid \text{ident}(L)$$

$$L \rightarrow E \mid L, E \mid \epsilon$$

### Question 1

[Solution n°61 p 86]

En utilisant les conventions de génération de code de la question 1.3, écrire les actions sémantiques correspondant aux appels de fonction.

*Note: On se contente ici de générer les appels de fonction sans se soucier de vérifier leur correction.*



Question 2

[*Solution n°62 p 88*]

Même question en adoptant les conventions de la question 4.

# TD 8 - Génération de code

IX

Pré-requis du TD 8	35
Exercice 1	35
Exercice 2	36
Exercice 3	36

## A. Pré-requis du TD 8

- Avoir suivi le cours de compilation sur la génération de code.

## B. Exercice 1

On souhaite générer du code pour évaluer des expressions sur une machine à registres (pas d'opération mémoire-registre). On considère l'expression suivante :  
 $(a + b) * (c + d) / (a - c) * (b + d) * e$

### Question 1

[Solution n°63 p 89]

Dessinez l'arbre abstrait de cette expression et calculez le nombre de registres nécessaires pour évaluer cette expression (on utilisera l'algorithme de Sethi et Ullman vu en cours). On supposera que toutes les variables sont à initialement placées en mémoire.

On suppose dans la suite que notre machine ne dispose que de deux registres.

### Question 2

[Solution n°64 p 92]

Calculer le nombre minimal d'opérations de spill nécessaires pour évaluer l'expression.

### Question 3

[Solution n°65 p 94]

On souhaite ne pas générer de code pour une expression dont les deux opérandes sont des constantes, i.e., c'est le compilateur lui-même qui doit calculer les sous-expressions constantes. Proposez une méthode permettant d'implémenter une telle optimisation dans un compilateur.

## C.Exercice 2

Reprendons la grammaire de la mini-calculatrice :

$$\begin{aligned} S &\mapsto I ; S \mid \epsilon \\ I &\mapsto \text{ident} := E \mid E \mid \epsilon \\ E &\mapsto E + T \mid E - T \mid T \\ T &\mapsto T * F \mid T / F \mid F \\ F &\mapsto \text{ident} \mid \text{const} \mid ( E ) \end{aligned}$$

On souhaite maintenant traduire un programme écrit dans ce langage de mini-calculatrice en un code cible trois adresses proche du C, soit une séquence d'instructions de la forme :

$x = y op z ;$

où  $x, y$  et  $z$  sont des identificateurs, des constantes ou des variables temporaires produites par le compilateur. Le terme  $op$  dénote un des opérateurs  $+, -, *, /$ .

Ainsi l'expression  $x + y * z$  pourra par exemple être traduite de la manière suivante :

$_t1 = y * z ;$

$_t2 = x + ,t1 ;$

### Question 1

[Solution n°66 p 97]

On considère dans un premier temps que l'on dispose :

- D'une fonction  $\text{new\_temp}$  qui fournit un nouveau nom de variable temporaire non utilisé.
- D'un attribut  $Code$  pour chaque symbole syntaxique non terminal.
- D'un attribut  $nom,es$  pour chaque symbole syntaxique non terminal qui contient le nom de la variable stockant le résultat.

Proposez des actions sémantiques pour les règles de la grammaire qui génèrent du code trois adresses.

### Question 2

[Solution n°67 p 98]

Dessinez l'arbre de dérivation et donnez le code généré pour l'entrée suivante :  
 $a := 3 ; (a + 5) * 2 ;$

## D.Exercice 3

Afin de traiter les boucles, on ajoute à la grammaire de la calculatrice la production suivante :

$$I \mapsto \text{for } \text{ident} := E \text{ to } E \{ S \}$$

On ajoute également dans le langage trois adresses les instructions suivantes :

- Le branchement inconditionnel  $gotoL$  qui a pour effet de faire exécuter ensuite l'instruction étiquetée par  $L$ .
- Le branchement conditionnel  $:if(xop)gotoL$ . Le terme  $op$  désigne un des opérateurs de comparaison  $<, >, <=, >=$ .

### Question 1

[Solution n°68 p 98]

Définir un modèle de traduction de la structure de contrôle for à l'aide des



branchements inconditionnels et conditionnels du langage trois adresses.

### Question 2

[Solution n°69 p 99]

Proposer une traduction dirigée par la syntaxe pour appliquer ce modèle de traduction et générer le code. On dispose:

- D'un attribut *Code* pour chaque symbole syntaxique non terminal.
- D'un attribut *nom<sub>r</sub>es* pour E qui contient le nom de la variable stockant le résultat.
- D'une fonction *new\_label* qui fournit une nouvelle étiquette non utilisée.

### Question 3

[Solution n°70 p 99]

Dessiner l'arbre de dérivation et donner le code généré pour :

```
a := 1 ;
for b := 5 + 3 to 10 {
    a := a + 1 ;
}
```

# TD 9 - Génération de code et optimisation.

X

Pré-requis du TD 9

39

Exercice 1

39

Exercice 2

40

## A. Pré-requis du TD 9

- Avoir compris le cours de compilation sur la génération de code.
- Avoir suivi le cours de compilation sur l'introduction à l'optimisation de code.

## B. Exercice 1

La grammaire de la mini-calculatrice est à présent étendue pour permettre l'utilisation de conditionnelles:

$$\begin{aligned} I &\mapsto \text{if}(B) \text{ then } \{S\} \\ I &\mapsto \text{if}(B) \text{ then } \{S\} \text{ else } \{S\} \\ B &\mapsto B \text{ or } B \mid B \text{ and } B \mid \text{not } B \\ B &\mapsto (B) \mid E \text{ op } E \mid \text{true} \mid \text{false} \end{aligned}$$

On suppose ce qui suit :

- le terminal *op* a un attribut *comp* qui sert à déterminer le type d'opérateur de comparaison utilisé parmi *<*, *>*, *==*, *!=*, etc.
- le non-terminal *B* possède les attributs suivants:
  - Deux étiquettes *B.true* et *B.false*.
  - Un code généré *B.code*. Ce code génère un branchement vers *B.true* si l'expression est vraie, et vers *B.false* dans le cas contraire.
- le non-terminal *E* a un attribut *Code* qui contient le code évaluant l'expression, et un attribut *nom\_var* qui contient le nom de la variable contenant le résultat de *E*.
- On dispose d'une fonction *nouvelle\_etiquette* qui fournit un nouveau nom d'étiquette non utilisé à chaque appel.

On désire traduire le code de la calculatrice en code trois adresses:

Question 1

[Solution n°71 p 100]

Proposer une traduction dirigée par la syntaxe qui génère le code des productions du type  $I \mapsto \dots$ . On ne soucie donc pas pour l'instant de la génération de  $B.code$ .

Question 2

[Solution n°72 p 101]

Considérons dans cette question la sous-grammaire associée aux productions des expressions booléennes  $B \mapsto \dots$ . Proposer une traduction dirigée par la syntaxe (par nécessité S-attribuée) qui génère du code pour les expressions booléennes. On souhaite que l'évaluation des expressions booléennes soit optimisée, c'est-à-dire que l'évaluation d'une expression ( $B1 \text{ and } B2$ ) ( resp. ( $B1 \text{ or } B2$ ) ) s'arrête si  $B1$  est évalué à faux ( resp. vrai ). A cet effet, reportez-vous au modèle de traduction dit "par branchements" tel vu au cours.

Question 3

[Solution n°73 p 103]

La traduction automatique de  $B \mapsto E_1 < E_2$  en code trois adresses produit la paire d'instructions:

*if* ( $E_1 < E_2$ ) *goto* ...  
*goto* ...

On pourrait se limiter à une seule instruction:

*if* ( $E_1 \geq E_2$ ) *goto* ...

et faire en sorte que l'exécution suive son cours si l'inégalité est vraie. En d'autres termes, nous ne générerons pas deux branchements, l'un vers une étiquette  $B.true$  et l'autre vers une étiquette  $B.false$ , mais uniquement un seul branchement vers l'une des deux étiquettes. Modifier en conséquence la traduction dirigée par la syntaxe de la question précédente.

## C.Exercice 2

On étudie dans cet exercice quelques optimisations sur du code trois adresses découpé en blocs de base. On considère le petit programme suivant en syntaxe mini-calculatrice:

```
for  i := 0 to 10 {  
    a := i + 1;  
    b := i + 1;  
    c := i + 2;  
    a + b;  
}
```

Question 1

[Solution n°74 p 104]

Traduire le programme en code trois adresses.

Question 2

[Solution n°75 p 105]

Découper ce code trois adresses en blocs de base.



Question 3

Représenter le graphe de flot de contrôle ainsi obtenu.

Question 4

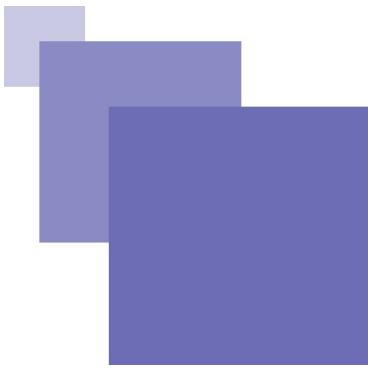
[*Solution n°76 p 105*]

Effectuer à la main une élimination de sous-expressions communes sur chacun des blocs.

Question 5

[*Solution n°77 p 105*]

Effectuer une élimination de code mort sur chacun des blocs.



# Solution des exercices rédactionnels

## > Solution n°1 (exercice p. 5)

### Code assembleur commenté

chaine :

```
.string "%d\n"
.globl main
.type main, @function
main :
    pushl %ebp          /* sauvegarde de ebp */
    movl %esp, %ebp     /* sauvegarde de esp */
    subl $24, %esp      /* sp -= 24 (allocation
                           par blocs) */
    movl $0, tab          /* tab[0] = 0 */
    movl $1, -4(%ebp)    /* i = 1 */
loop1 :
    cmpl $99, -4(%ebp)  /* i < 100 ? */
    jle body1            /* oui -> .body1 */
    jmp skip1
body1 :
    movl -4(%ebp), %edx  /* edx = i */
    leal 0(%edx, 4), %eax /* eax = 4 * i */
    movl $tab, %edx
    movl $1, (%eax, %edx) /* tab[i] = 1 */
```

```

incl -4(%ebp)          /* i ++ */
jmp loop1
skip1 :
    movl $2,-4(%ebp)    /* i = 2 */
loop2 :
    cmpl $9,-4(%ebp)    /* i < 10 ? */
    jle body2            /* oui -> body2 */
    jmp skip2
body2 :
    movl -4(%ebp),%eax  /* eax = i */
    movl %eax,%edx
    leal (%edx,%eax),%ecx /* ecx = i + i */
    movl %ecx,-8(%ebp)   /* j = i + i */
loop3 :
    cmpl $99,-8(%ebp)   /* j < 100 ? */
    jle body3            /* oui -> body 3 */
    jmp skip3
body3 :
    movl -8(%ebp),%edx  /* edx = j */
    leal 0(%edx,4),%eax  /* eax = 4 * j */
    movl $tab,%edx
    movl $0,(%eax,%edx)  /* tab[j] = 0 */
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)   /* j += i */
    jmp loop3
skip3 :
    incl -4(%ebp)        /* i ++ */
    jmp loop2
skip2 :
    movl $0,-4(%ebp)     /* i = 0 */
loop4 :
    cmpl $99,-4(%ebp)   /* i < 100 ? */
    jle body4            /* oui -> body4 */
    jmp skip4
body4 :
    movl -4(%ebp),%edx  /* edx = i */
    leal 0(%edx,4),%eax  /* eax = 4 * i */
    movl $tab,%edx
    cmpl $1,(%eax,%edx) /* tab[i] = 1 ? */
    jne else              /* non -> else */
    addl $-8,%esp         /* sp -= 8 */
    movl -4(%ebp),%eax
    pushl %eax            /* empile i */

```

```

pushl $chaine    /* empile "%d\n" */
call printf      /* appelle printf */
addl $16,%esp   /* sp += 16 */

else :
    incl -4(%ebp)  /* i++ */
    jmp loop4

skip4 :
    leave           /* équivalent à :
                      movl %ebp,%esp ;
                      popl %ebp */
    ret
.comm  tab,400,32

```

## Programme C source

```

#define N 100
#define SN 10          /* sqrt(n) */
int tab [N] ;

void main (void) {
    int i, j ;

    tab [0] = 0 ;      /* 0 est non premier */
    for (i = 1 ; i < N ; i++)
        tab [i] = 1 ;  /* ils sont tous premiers par défaut */
    for (i = 2 ; i < SN ; i++) {
        j = i + i ;
        while (j < N) {
            tab [j] = 0 ; /* les multiples de i ne sont pas premiers */
            j += i ;
        }
    }
    for (i = 0 ; i < N ; i++) {
        if (tab [i] == 1)
            printf ("%d\n", i) ;
    }
}

```

> **Solution n°2** (exercice p. 7)

```
.globl main
.type    main, @function
main :
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp          /* réserve 24 octets (= 6 * 4) sur la pile */
    movl -4(%ebp), %eax
    movl %eax, -8(%ebp)     /* j = i */
    movl $5, -4(%ebp)       /* i = 5 */
    leave
    ret
```

> Solution n°3 (exercice p. 8)

```
.globl main
.type    main, @function
main :
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl $0, -8(%ebp)        /* s = 0 */
    movl $0, -4(%ebp)        /* i = 0 */
loop :
    cmpl $99, -4(%ebp)      /* i < 100 ? */
    jle body
    jmp skip
body :
    movl -4(%ebp), %eax
    addl %eax, -8(%ebp)     /* s += i */
    incl -4(%ebp)           /* i++ */
    jmp loop
skip :
    leave
    ret
```

> Solution n°4 (exercice p. 8)

```

.globl main
.type main, @function
main :
    pushl %ebp
    movl %esp, %ebp
    subl $820, %esp      /* réserve 10 * 10 * 4
                           + 10 * 10 * 4 + 4 + 4 * 4 */
    pushl %ebx
    movl $1, -4(%ebp)
.L3 :
    cmpl $9, -4(%ebp)   /* i < 10 ? */
    jle .L6
    jmp .L4
.L6 :
    movl -4(%ebp), %eax
    movl %eax, %edx
    sall $2, %edx
    addl %eax, %edx
    addl %edx, %edx
    addl %eax, %edx

    leal 0(%edx, 4), %eax
    leal -404(%ebp), %edx
    movl -4(%ebp), %ebx
    movl %ebx, %ecx
    sall $2, %ecx
    addl %ebx, %ecx
    addl %ecx, %ecx
    addl %ebx, %ecx
    leal 0(%ecx, 4), %ebx
    leal -804(%ebp), %ecx
    movl (%ebx, %ecx), %ebx

    movl %ebx, (%eax, %edx)
    incl -4(%ebp)
    jmp .L3
    .p2align 4,,7
.L4 :
    movl -824(%ebp), %ebx
    leave
    ret

```

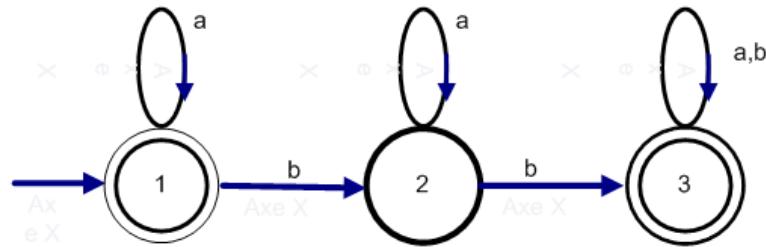
### > Solution n°5 (exercice p. 8)

Toutes les éléments du tableau sont contigus en mémoire => un tableau à trois dimensions représente un cube.

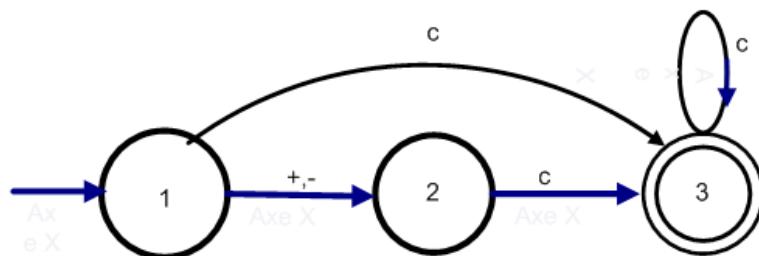
Si le tableau est rangé en lignes, cela veut dire que les éléments contigus sont tab[0][0][0], tab[0][0][1], tab[0][0][2], .... tab[0][0][n3-1], tab[0][1][0], tab[0][1][1], ....

adresse de l'élément  $\text{tab}[i][j][k] = (\text{n2} * \text{n3} * i + \text{n3} * j + k) * \text{sizeof( struct t )}$

**> Solution n°6** (exercice p. 9)

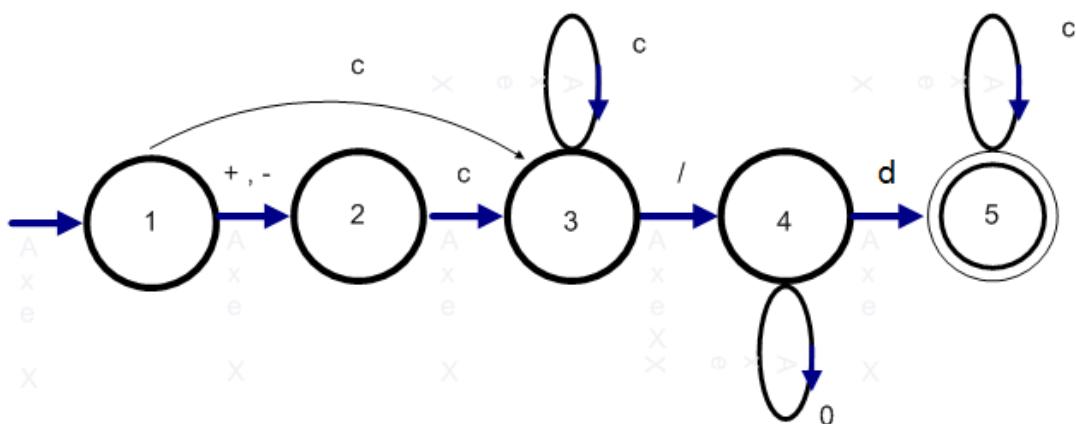


**> Solution n°7** (exercice p. 9)



$c$  dénote est un chiffre quelconque de l'alphabet  $A$

**> Solution n°8** (exercice p. 9)

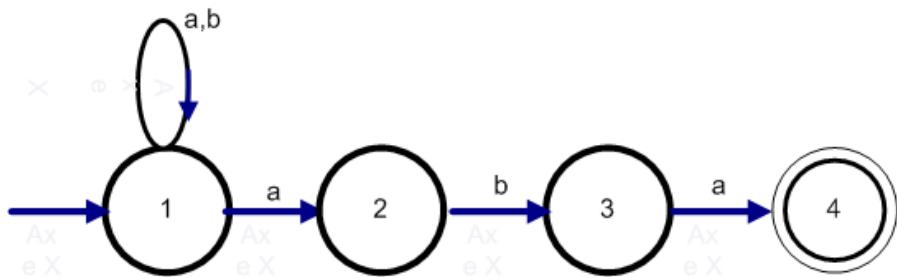


$c$  dénote est un chiffre quelconque de l'alphabet  $A$ , et  $d$  un chiffre différent de 0

**> Solution n°9** (exercice p. 9)

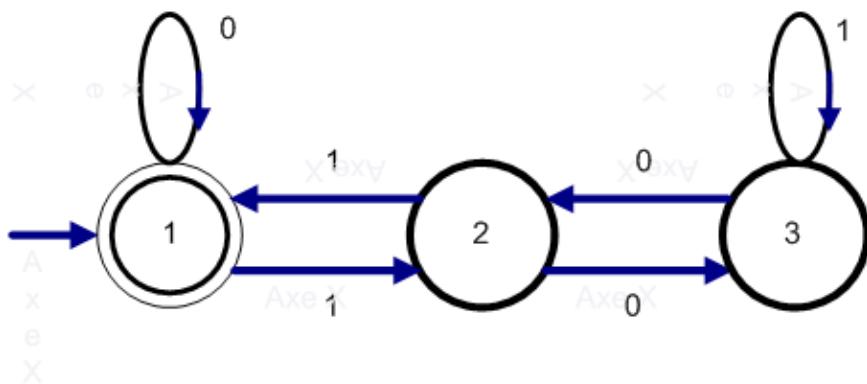


Oui, il est reconnu par:

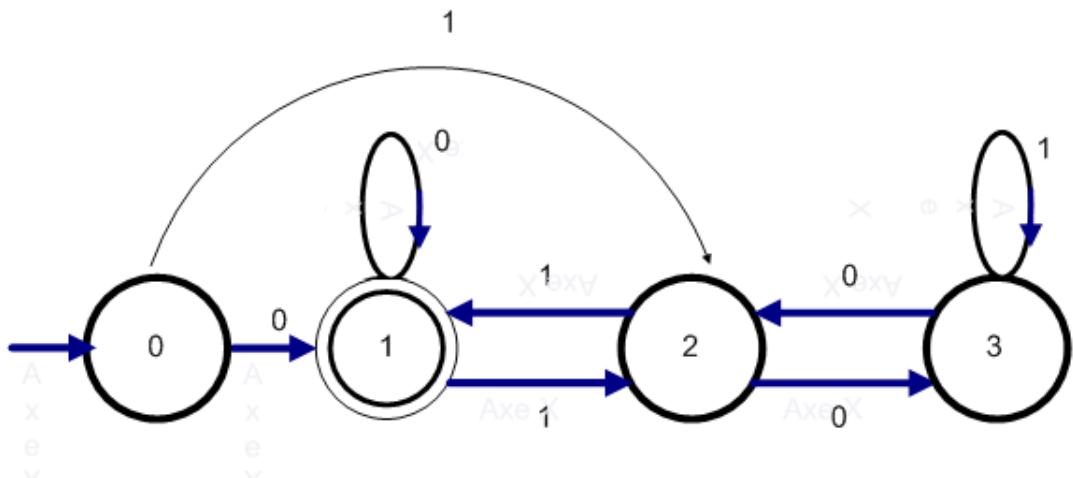


### > Solution n°10 (exercice p. 9)

Oui, il est reconnu par:



RQ: Cet automate reconnaît également le mot vide. Pour l'écartier, il suffit de considérer l'automate suivant:



### > Solution n°11 (exercice p. 9)

On crée un nouvel état  $p$  (état poubelle). Puis on ajoute des flèches de la forme  $(q, a, p)$ , pour chaque état  $q$  et chaque lettre  $a$  qui ne sont pas état de départ et étiquette d'une même flèche de  $X$ .

De plus, on ajoute  $\text{Card}(A)$  flèches de la forme  $(p, a, p)$ , pour chaque lettre  $a$  de  $A$ . Le nouvel automate  $X'$  est complet et reconnaît le même langage que  $X$ .

> **Solution n°12** (exercice p. 10)

Oui.  $L = \{a\}^* \cup \{a\}^*b\{a\}^*b\{a, b\}^*$

> **Solution n°13** (exercice p. 10)

$L = \{+, -, \epsilon\}\{0, 1, \dots, 9\}^+$ .

> **Solution n°14** (exercice p. 10)

$L = \{+, -, \epsilon\}\{0, 1, \dots, 9\}^+ / 0^*\{1, \dots, 9\}\{0, 1, \dots, 9\}^+$

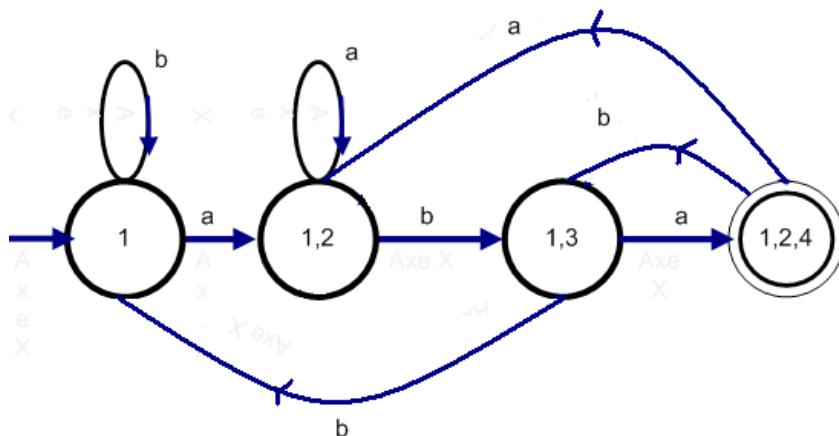
> **Solution n°15** (exercice p. 10)

Oui.  $L = \{a, b\}^*aba$ .

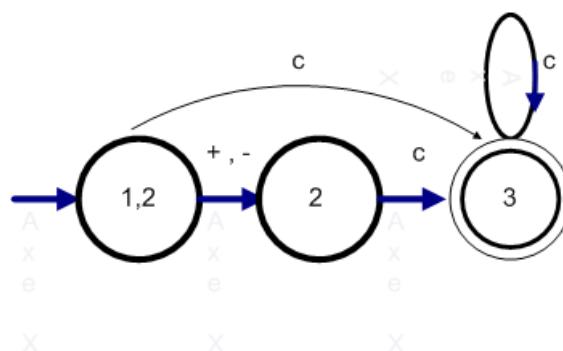
> **Solution n°16** (exercice p. 10)

$a^+b^* \cap \{a, b\}^*ba^* = a^+b^+$ .

> **Solution n°17** (exercice p. 10)



> **Solution n°18** (exercice p. 10)



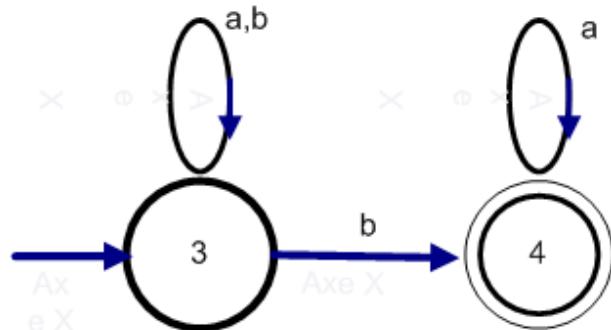
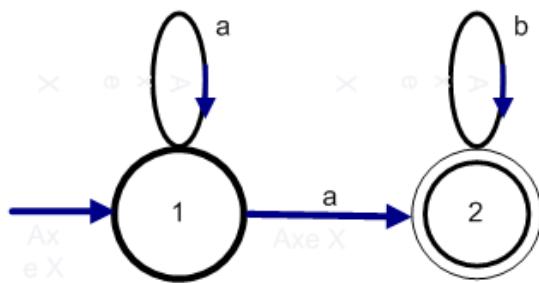
### > Solution n°19 (exercice p. 11)

$L_1$  et  $L_2$  sont réguliers, donc reconnaissables (Th. de Kleene) par  $X_1 = (Q_1, I_1, T_1, F_1)$ , et  $X_2 = (Q_2, I_2, T_2, F_2)$ .

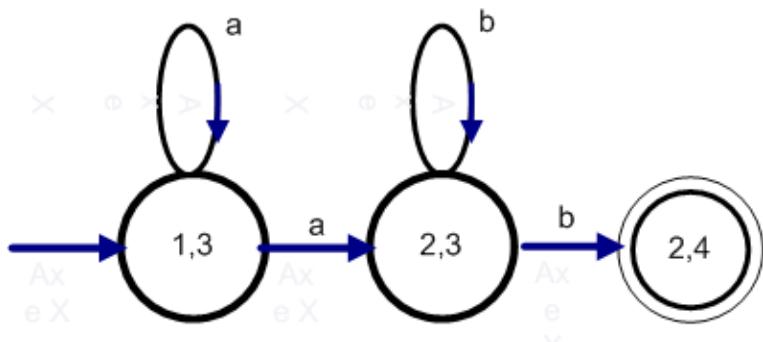
$\text{Intersection}(X_1, X_2) = (Q, I, T, F)$ , avec:

- $Q = Q_1 \times Q_2$ .
- $I = I_1 \times I_2$ .
- $T = T_1 \times T_2$ .
- $F = \{(q_1, q_2), a, (p_1, p_2)) \times Q_1 \times Q_2 \times A \times Q_1 \times Q_2 / (q_1, a, p_1) \in F_1 \text{ et } (q_2, a, p_2) \in F_2\}$

Les automates  $X_1$  et  $X_2$  reconnaissant  $a^+b^*$  et  $\{a, b\}^*ba^*$  sont respectivement:



$\text{Intersection}(X_1, X_2)$  est:



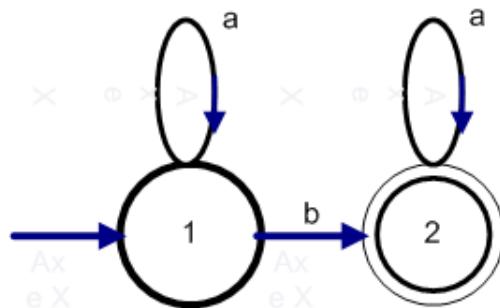
### > Solution n°20 (exercice p. 11)

$L_1$  est régulier, donc reconnaissable (Th. De Kleene) par  $X_1 = (Q_1, I_1, T_1, F_1)$ .

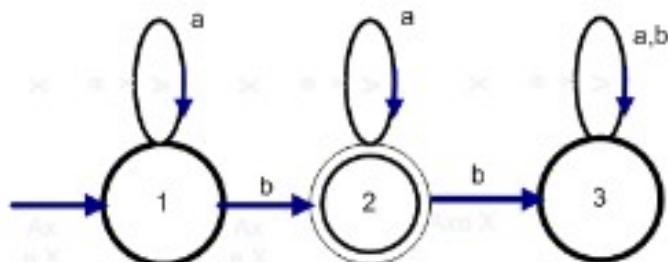
- On considère  $Y = \text{Déterministe}(X_1)$  l'automate déterministe associé à  $X_1$ , et  $Z = \text{Complet}(Y)$  l'automate complet de  $Z$  (cf. exercice 1.5)
- Soit  $Z = (Q, I, T, F)$ .

On a:  $\text{Complémentaire}(X_1) = (Q, I, Q \setminus T, F)$ ,

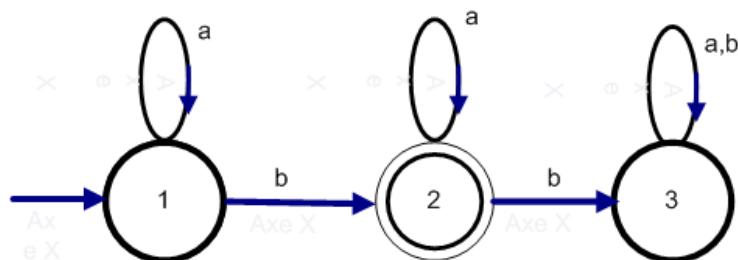
L'automate  $X_1$  reconnaissant  $a^*ba^*$  est:



Il est déterministe. Son  $\text{Complet}(X_1)$  est:



Son complémentaire est:

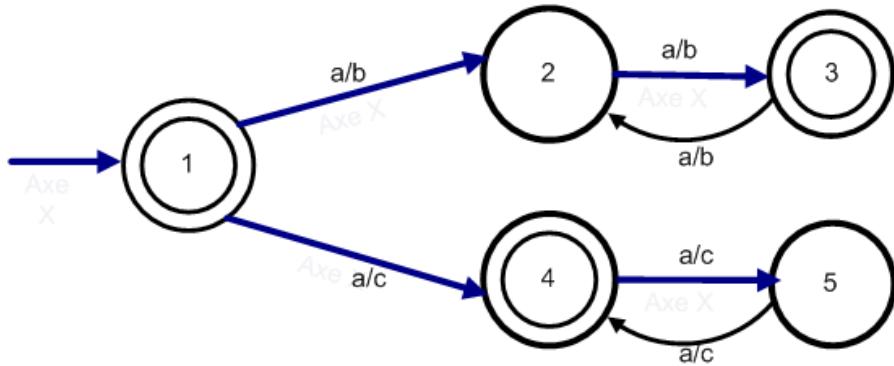


### > Solution n°21 (exercice p. 11)

$L_1 \cap L_2$  et  $A^* \setminus L_1$  sont donc réguliers, car reconnaissables (Th. de Kleene). Il peuvent donc se réécrire en utilisant uniquement les trois opérations régulières.

### > Solution n°22 (exercice p. 11)





## &gt; Solution n°23 (exercice p. 13)

Premiers	
$S$	$\{a, b, c\}$
$A$	$\{\epsilon, c, b\}$
$B$	$\{b\}$
$C$	$\{c, \epsilon\}$
$AaB$	$\{a, b, c\}$
$CB$	$\{c, b\}$
$Bb$	$\{b\}$

Suivants	
$S$	$\{\$\}$
$A$	$\{a\}$
$B$	$\{b, \$, a\}$
$C$	$\{b\}$

## &gt; Solution n°24 (exercice p. 13)

	$a$	$b$	$c$	$\$$
$S$	$S \mapsto AaB$	$S \mapsto AaB$	$S \mapsto AaB$	
$A$	$A \mapsto \epsilon$	$A \mapsto CB$	$A \mapsto CB$	
$B$		$B \mapsto b$		
$C$		$C \mapsto \epsilon$	$C \mapsto c$	

Cette grammaire n'est pas LL(1) puisque la table d'analyse possède une cellule contenant plus d'une règle.

## &gt; Solution n°25 (exercice p. 13)

Pour  $5+3*2$  ;

$$\begin{aligned}
 S &\Rightarrow E ; S \Rightarrow E + T ; S \Rightarrow T + T ; S \Rightarrow F + T ; S \Rightarrow \text{const} + T ; S \\
 &\Rightarrow \text{const} + T * F ; S \Rightarrow \text{const} + F * F ; S \Rightarrow \text{const} + \text{const} * F ; S \\
 &\Rightarrow \text{const} + \text{const} * \text{const} ; S \Rightarrow \text{const} + \text{const} * \text{const} ;
 \end{aligned}$$

### Pour 3 ;2/3\*(1-3)

$$\begin{aligned}
 S &\Rightarrow E ; S \Rightarrow T ; S \Rightarrow F ; S \Rightarrow \text{const} ; S \Rightarrow \text{const} ; E ; S \Rightarrow \\
 &\text{const} ; T ; S \Rightarrow \text{const} ; T * F ; S \Rightarrow \text{const} ; T / F * F ; S \Rightarrow \text{const} ; F / F * F ; S \\
 &\Rightarrow \text{const} ; \text{const} / F * F ; S \Rightarrow \text{const} ; \text{const} / \text{const} * F ; S \\
 &\Rightarrow \text{const} ; \text{const} / \text{const} * (E) ; S \Rightarrow \text{const} ; \text{const} / \text{const} * (E - T) ; S \\
 &\Rightarrow \text{const} ; \text{const} / \text{const} * (T - T) ; S \Rightarrow \text{const} ; \text{const} / \text{const} * (F - T) ; S \\
 &\Rightarrow \text{const} ; \text{const} / \text{const} * (\text{const} - T) ; S \Rightarrow \text{const} ; \text{const} / \text{const} * (\text{const} - \\
 &F) ; S \Rightarrow \text{const} ; \text{const} / \text{const} * (\text{const} - \text{const}) ; S \\
 &\Rightarrow \text{const} ; \text{const} / \text{const} * (\text{const} - \text{const}) ;
 \end{aligned}$$

### > Solution n°26 (exercice p. 14)

La règle  $E \mapsto E + T$  montre que cette grammaire est récursive à gauche. La variable  $E$  est à la fois en tête de règle et du choix de cette règle.

Elimination des récursivités à gauche :

De manière générale :

$N \mapsto Na|b$  devient  $N \mapsto bN'$  et  $N' \mapsto aN'|\epsilon$

$$\begin{aligned}
 S &\mapsto E ; S | \epsilon \\
 E &\mapsto T E2 \\
 E2 &\mapsto + T E2 | - T E2 | \epsilon \\
 T &\mapsto F T2 \\
 T2 &\mapsto * F T2 | / F T2 | \epsilon \\
 F &\mapsto \text{const} | ( E )
 \end{aligned}$$

### > Solution n°27 (exercice p. 14)

On utilise la grammaire non récursive à gauche puisque aucune grammaire récursive à gauche ne peut être LL(1).

### > Solution n°28 (exercice p. 14)

On réécrit les règles sous la forme :



Règle 11  $S \rightarrow E ; S$   
Règle 12  $S \rightarrow \epsilon$   
Règle 2  $E \rightarrow T E2$   
Règle 31  $E2 \rightarrow + T E2$   
Règle 32  $E2 \rightarrow - T E2$   
Règle 33  $E2 \rightarrow \epsilon$   
Règle 4  $T \rightarrow F T2$   
Règle 51  $T2 \rightarrow * F T2$   
Règle 52  $T2 \rightarrow / F T2$   
Règle 53  $T2 \rightarrow \epsilon$   
Règle 61  $F \rightarrow \text{const}$   
Règle 62  $F \rightarrow ( E )$

Premiers	
$S$	{const, (, $\epsilon$ )}
$E$	{const, ()}
$T$	{const, ()}
$F$	{const, ()}
$E2$	{+, -, $\epsilon$ }
$T2$	{*, /, $\epsilon$ }

Suivants	
$S$	{\$}
$E2$	{; , ) }
$T2$	{+ , - , ; , ) }

Table d'analyse :

	;	+	-	*	/	const	(	)	\$
$S$						Règle 11	Règle 11		Règle 12
$E$						Règle 2	Règle 2		
$T$						Règle 4	Règle 4		
$F$						Règle 61	Règle 62		
$E2$	Règle 33	Règle 31	Règle 32						Règle 33
$T2$	Règle 53	Règle 53	Règle 53	Règle 51	Règle 52				Règle 53

Analyse de  $3 * (1 + 2) / (1);$

## TD 9 - Génération de code et optimisation.

Entrée	Pile (fond à droite)	Action	Sortie
3*(1+2)/(1);\$	S\$	Production	11
3*(1+2)/(1);\$	E ; S\$	Production	11 2
3*(1+2)/(1);\$	T E2 ; S\$	Production	11 2 4
3*(1+2)/(1);\$	F T2 E2 ; S\$	Production	11 2 4 61
3*(1+2)/(1);\$	const T2 E2 ; S\$	Matching	11 2 4 61
*(1+2)/(1);\$	T2 E2 ; S\$	Production	11 2 4 61 51
*(1+2)/(1);\$	* F T2 E2 ; S\$	Matching	11 2 4 61 51
(1+2)/(1);\$	F T2 E2 ; S\$	Production	11 2 4 61 51 62
(1+2)/(1);\$	( E ) T2 E2 ; S\$	Matching	11 2 4 61 51 62
1+2)/(1);\$	E ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2
1+2)/(1);\$	T E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4
1+2)/(1);\$	F T2 E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61
1+2)/(1);\$	const T2 E2 ) T2 E2 ; S\$	Matching	11 2 4 61 51 62 2 4 61
+2)/(1);\$	T2 E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53
+2)/(1);\$	E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31
+2)/(1);\$	+ T E2 ) T2 E2 ; S\$	Matching	11 2 4 61 51 62 2 4 61 53 31
2)/(1);\$	T E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31 4
2)/(1);\$	F T2 E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31 4 61
2)/(1);\$	const T2 E2 ) T2 E2 ; S\$	Matching	11 2 4 61 51 62 2 4 61 53 31 4 61
)/(1);\$	T2 E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31 4 61 53
)/(1);\$	E2 ) T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31 4 61 53 33
)/(1);\$	) T2 E2 ; S\$	Matching	11 2 4 61 51 62 2 4 61 53 31 4 61 53 33
/(1);\$	T2 E2 ; S\$	Production	11 2 4 61 51 62 2 4 61 53 31 4 61 53 33 52
/(1);\$	/ F T2 E2 ; S\$	Matching	11 2 4 61 51 62 2 4 61 53 31 4 61 53 33 52
(1);\$	F T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62
(1);\$	( E ) T2 E2 ; S\$	Matching	... 2 4 61 53 31 4 61 53 33 52 62
1);\$	E ) T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2
1);\$	T E2 ) T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2
1);\$	F T2 E2 ) T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2 61
1);\$	const T2 E2 ) T2 E2 ; S\$	Matching	... 2 4 61 53 31 4 61 53 33 52 62 2 61
);\$	T2 E2 ) T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53
);\$	E2 ) T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33
);\$	) T2 E2 ; S\$	Matching	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33
;\$	T2 E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33 53
;\$	E2 ; S\$	Production	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33 53 33
;\$	; S\$	Matching	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33 53 33
\$	S\$	Acceptation	... 2 4 61 53 31 4 61 53 33 52 62 2 61 53 33 53 33

## > Solution n°29 (exercice p. 14)

Premiers		Suivants	
< inst >	{IF, ID}	< inst >	{ELSE, \$}
< expression >	{ID}	< expression >	{THEN}
< else – inst >	{ELSE, ε}	< else – inst >	{ELSE, \$}

Table d'analyse :

	IF	ID	THEN	:=	ELSE	\$
< inst >	Règle 1	Règle 2				
< expression >		Règle 5				
< else – inst >					Règles 3 et 4	Règle 4

## > Solution n°30 (exercice p. 14)

On constate un conflit Premier, Suivant pour la variable < else – inst > à la

rencontre du symbole *ELSE*. On priviléie la règle 3 qui consiste à produire *ELSE < instr >* ce qui associe *ELSE* au *THEN* le plus proche.

### > Solution n°31 (exercice p. 14)

Suivant l'analyse précédente on devra la considérer comme :

*IF x THEN (IF y THEN i := j ELSE i := k)*

et non comme :

*IF x THEN (IF y THEN i := j) ELSE i := k*

Analyse :

Entrée	Pile (fond à droite)	Action	Sortie
IF x THEN IF y THEN i := j ELSE i := k\$			
IF x THEN IF y THEN i := j ELSE i := k\$	< inst >\$	Production	1
x THEN IF y THEN i := j ELSE i := k\$	< expression > THEN < inst > < else - inst >\$	Matching	1
x THEN IF y THEN i := j ELSE i := k\$	< expression > THEN < inst > < else - inst >\$	Production	1 5
THEN IF y THEN i := j ELSE i := k\$	ID THEN < inst > < else - inst >\$	Matching	1 5
THEN IF y THEN i := j ELSE i := k\$	THEN < inst > < else - inst >\$	Matching	1 5
IF y THEN i := j ELSE i := k\$	< inst > < else - inst >\$	Production	1 5 1
IF y THEN i := j ELSE i := k\$	IF < expression > THEN < inst > < else - inst > < else - inst >\$	Matching	1 5 1
y THEN i := j ELSE i := k\$	< expression > THEN < inst > < else - inst > < else - inst >\$	Production	1 5 1 5
y THEN i := j ELSE i := k\$	ID THEN < inst > < else - inst > < else - inst >\$	Matching	1 5 1 5
THEN i := j ELSE i := k\$	THEN < inst > < else - inst > < else - inst >\$	Matching	1 5 1 5 2
i := j ELSE i := k\$	< inst > < else - inst > < else - inst >\$	Production	1 5 1 5 2
i := j ELSE i := k\$	ID := ID < else - inst > < else - inst >\$	Matching	1 5 1 5 2
:= j ELSE i := k\$	:= ID < else - inst > < else - inst >\$	Matching	1 5 1 5 2
j ELSE i := k\$	ID < else - inst > < else - inst >\$	Matching	1 5 1 5 2
ELSE i := k\$	< else - inst > < else - inst >\$	Production	1 5 1 5 2 3 (choix)
ELSE i := k\$	ELSE < inst > < else - inst >\$	Matching	1 5 1 5 2 3
i := k\$	< inst > < else - inst >\$	Production	1 5 1 5 2 3 2
i := k\$	ID := ID < else - inst > \$	Matching	1 5 1 5 2 3 2
k\$	ID < else - inst > \$	Matching	1 5 1 5 2 3 2
\$	< else - inst > \$	Production	1 5 1 5 2 3 2 4
\$	\$	Acceptation	1 5 1 5 2 3 2 4

### > Solution n°32 (exercice p. 15)

Items LR(0) et états :

I0 =	{ S' → . < inst > \$, < inst > → . IF < expression > THEN < inst > < else - inst >, < inst > → . ID := ID }
I1 = δ(I0, inst <sub>c</sub> ) =	{ S' → < inst > . \$ }
I2 = δ(I0, IF) =	{ < inst > → IF, < expression > THEN < inst > < else - inst >, < expression > → . ID }
I3 = δ(I0, ID) =	{ < inst > → ID . := ID }
I4 = δ(I1, \$) =	{ S' → < inst > \$ . }
I5 = δ(I2, < expression >) =	{ < inst > → IF < expression > . THEN < inst > < else - inst > }
I6 = δ(I2, ID) =	{ < expression > → ID . }
I7 = δ(I3, :=) =	{ < inst > → ID . := ID }
I8 = δ(I5, THEN) =	{ < inst > → IF < expression > THEN . < inst > < else - inst >, < inst > → . ID := ID }
I9 = δ(I7, ID) =	{ < inst > → ID . }
I10 = δ(I8, < inst >) =	{ < inst > → IF < expression > THEN < inst > . < else - inst >, < else - inst > → . ELSE < inst >, < else - inst > → . }
δ(I8, IF) =	{ < inst > → IF . < expression > THEN < inst > < else - inst >, < expression > → . ID }
δ(I8, ID) =	= I2
δ(I8, :=) =	{ < inst > → ID . := ID }
I11 = δ(I10, < else - inst >) =	{ < inst > → IF < expression > THEN < inst > < else - inst > . }
I12 = δ(I10, ELSE) =	{ < else - inst > → ELSE . < inst >, < inst > → . IF < expression > THEN < inst > < else - inst >, < inst > → . ID := ID }
I13 = δ(I12, inst <sub>c</sub> ) =	{ < else - inst > → ELSE < inst > . }
δ(I12, ID) =	{ < inst > → ID . := ID }
	= I3

L'état I10 présente un conflit décaler - réduire ; la grammaire n'est pas LR(0).

### > Solution n°33 (exercice p. 15)

Premiers	
$<inst>$	$\{IF, ID\}$
$<else-inst>$	$\{ELSE, \epsilon\}$
$<expression>$	$\{ID\}$
$<inst><else-inst>$	$\{IF, ID\}$

Suivants	
$<inst>$	$\{\$, ELSE\}$
$<else-inst>$	$\{\$, ELSE\}$
$<expression>$	$\{THEN\}$

En présence du symbole ELSE, l'état 10 présente :

- un décalage vers l'état 12 ( $I12 = \delta(I10, ELSE)$ ),
- et une réduction par la règle  $<else-inst> \rightarrow \epsilon$  car  $ELSE \in \text{SUIVANTS}(<else-inst>)$ .

La grammaire n'est pas SLR(1).

### > Solution n°34 (exercice p. 15)

On choisira de privilégier le décalage en présence du symbole ELSE ce qui attache le ELSE au THEN le plus récent.

### > Solution n°35 (exercice p. 15)

$$\begin{aligned}
 I0 &= \{ [S' \rightarrow . S \$, \epsilon], [S \rightarrow . A a, \$], [S \rightarrow . b A c, \$] \\
 &\quad [S \rightarrow . d c, \$], [S \rightarrow . b d a, \$], [A \rightarrow . d, a] \} &= I1 \\
 \delta(I0, S) &= \{ [S' \rightarrow S . \$, \epsilon] \} &= I1 \\
 \delta(I0, A) &= \{ [S \rightarrow A . a, \$] \} &= I2 \\
 \delta(I0, b) &= \{ [S \rightarrow b . A c, \$], [S \rightarrow b . d a, \$], [A \rightarrow . d, c] \} &= I3 \\
 \delta(I0, d) &= \{ [S \rightarrow d . c, \$], [A \rightarrow d . a] \} &= I4 \\
 \delta(I1, \$) &= \{ [S' \rightarrow S \$ . , \epsilon] \} &= I5 \\
 \delta(I2, a) &= \{ [S \rightarrow A a . , \$] \} &= I6 \\
 \delta(I3, A) &= \{ [S \rightarrow b A . c, \$] \} &= I7 \\
 \delta(I3, d) &= \{ [S \rightarrow b d . a, \$], [A \rightarrow d . c] \} &= I8 \\
 \delta(I4, c) &= \{ [S \rightarrow d c . , \$] \} &= I9 \\
 \delta(I7, c) &= \{ [S \rightarrow b A c . , \$] \} &= I10 \\
 \delta(I8, a) &= \{ [S \rightarrow b d a . , \$] \} &= I11
 \end{aligned}$$

Les états ne présentent pas de conflits. La grammaire est LR(1).

Aucun état ne peut fusionner avec un autre. La grammaire est LALR(1).

On construit les items LR(0) pour l'étude SLR(1).



I0=	$\{S' \rightarrow . S \$, S \rightarrow . A a, S \rightarrow . b A c$	
	$S \rightarrow . d c, S \rightarrow . b d a, A \rightarrow . d\}$	
$\delta(I0, S) =$	$\{S' \rightarrow S . \$\}$	=I1
$\delta(I0, A) =$	$\{S \rightarrow A . a\}$	=I2
$\delta(I0, b) =$	$\{S \rightarrow b . A c, S \rightarrow b . d a, A \rightarrow . d\}$	=I3
$\delta(I0, d) =$	$\{S \rightarrow d . c, A \rightarrow d .\}$	=I4
$\delta(I1, \$) =$	$\{S' \rightarrow S \$ .\}$	=I5
$\delta(I2, a) =$	$\{S \rightarrow A a .\}$	=I6
$\delta(I3, A) =$	$\{S \rightarrow b A . c\}$	=I7
$\delta(I3, d) =$	$\{S \rightarrow b d . a, A \rightarrow d .\}$	=I8
$\delta(I4, c) =$	$\{S \rightarrow d c .\}$	=I9
$\delta(I7, c) =$	$\{S \rightarrow b A c .\}$	=I10
$\delta(I8, a) =$	$\{S \rightarrow b d a .\}$	=I11

$Suivant(S) = \{\$\}$  et  $Suivant(A) = \{a, c\}$

L'état I4 présente un conflit décaler-réduire :

$\delta(I4, c) = I9$  indique un décalage vers I9 pour c.

$A \rightarrow d .$  indique une réduction par la règle  $A \rightarrow d$  pour chaque suivant de A et en particulier c.

De même l'état I8 présente le même type de conflit.

La grammaire n'est pas SLR(1).

### > Solution n°36 (exercice p. 17)

#### fichier lex.l

```
%{
    #include <stdio.h>
    #include "y.tab.h"
%}
C [0-9]
%%
{C}+ return NB; /* Le token NB est une suite de chiffres */
[ \n\t]; /* Les espaces et tabulation ne sont pas significatifs */
.return yytext[0]; /* Tout le reste est renvoyé au parser */
%%
```

#### fichier yacc.y

## TD 9 - Génération de code et optimisation.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token NB

%%

S :
E {printf("chaîne acceptée");}

E :
E' +' T
| T

T :
T' *' F
| F

F :
'( E ')
| NB

%%

int yyerror( char * s ) {
    fprintf( stderr, "%s\n", s );
    exit(1); /* le programme s'arrête lors d'une erreur de syntaxe */
}

int main() {
    while (1)
        yyparse();
}
```

### > Solution n°37 (exercice p. 17)

#### fichier lex.l

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval; /* La variable qui contient la valeur (attribut) du token */
%}
C [0 - 9]
%%
{C} + {
    sscanf (yytext, "%d", &yylval); /* convertir la chaîne de texte en entier */
    return NB; /* retourner le token */
}
[ \n\t]; /* Les espaces et tabulation ne sont pas significatifs */
.return yytext[0]; /* Tout le reste est renvoyé au parser */
%%
```

#### fichier yacc.y

```
%{
#include <stdio.h>

%}

%token NB

%%%
S :
E {printf("résultat = %d\n", $1);}

E :
E +' T {$$ = $1 + $3;}
| T {$$ = $1;};

T :
T '*' F {$$ = $1 * $3;}
| F {$$ = $1;};

F :
'(' E ')' {$$ = $2;}
| NB {$$ = $1;};

%%%
```

### > Solution n°38 (exercice p. 18)

```
grammar Expression;
options {
    language = Java;
}

@lexer :: header {
    package expression;
}

@header {
    package expression;
}

INT : '0'..'9'+ ;
WS : (' '|'\t') + {skip();} ;

expr : terme ('+' terme)*
;
terme : facteur ('*' facteur)*
;
facteur :
n = INT {System.out.println("Entier : " + $n.text);}
| '(' expr ')'
;
```

### > Solution n°39 (exercice p. 18)

```

grammar Expression;
options {
    language = Java;
}

@lexer :: header {
    package expression;
}

@header {
    package expression;
}

INT : '0'..'9'+ ;
WS : (' '|'\t') + {skip();} ;

expr returns [double r = 0] :
    t = terme {r = t;} ('+' t = terme {r += t;}) *
    ;
terme returns [double r = 0] :
    v = facteur {r = v;} ('*' v = facteur {r *= v;}) *
    ;
facteur returns [double r = 0] :
    n = INT {r = Double.parseDouble($n.getText());}
    | '(' v = expr ')' {r = v;}
    ;

```

### > Solution n°40 (exercice p. 18)

```

public static void calcul() {
    try {
        FileInputStream fis = new FileInputStream("../expression.prg");
        ANTLRInputStream input = new ANTLRInputStream(fis);
        ExpressionLexer lexer = new ExpressionLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExpressionParser parser = new ExpressionParser(tokens);
        System.out.println(parser.expr());
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

```

### > Solution n°41 (exercice p. 19)

Table des symboles : on peut penser à plusieurs fonctions d'accès et de maj de la table des symboles

- *symbole chercher – symbole (string nom)*  
cherche un symbole et l'ajoute s'il n'est pas trouvé
- *void ecrire – valeur (symbole s, integer v)*  
associe la valeur v au symbole, et marque le symbole comme étant initialisé.
- *integer lire – valeur (symbole s)*  
lit la valeur associée au symbole.
- *boolean est – initialisee (symbole s)*



indique si la variable est déjà initialisée ou non.

### > Solution n°42 (exercice p. 19)

## Actions sémantiques

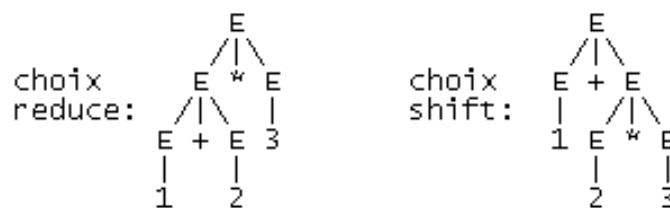
```

 $F \mapsto ident$ 
    /* ident.entre'e est fournie par l'AL */
    if (est-initialisee (ident.entre'e))
        F.val := lire-valeur (ident.entre'e)
    else
        erreur ("symbole %s non initialisé", ident.entre'e)
 $F \mapsto const$ 
    F.val := const.val (fournie par l'AL)
 $F \mapsto (E)$ 
    F.val := E.val
 $T \mapsto T * F | T / F$ 
    T1.val := T2.val{*/} F.val
 $T \mapsto F$ 
    T.val := F.val
 $E \mapsto E + T | E - T$ 
    E1.val := E2.val{+-} T.val
 $E \mapsto T$ 
    E.val := T.val
 $I \mapsto \epsilon$ 
    /* rien */
 $I \mapsto E$ 
    printf("%d\n", E.val)
 $I \mapsto letident := E$ 
    ecrire-valeur (ident.entre'e, E.val)
 $S \mapsto I ; S$ 
    /* rien */
 $S \mapsto \epsilon$ 
    /* rien */

```

### > Solution n°43 (exercice p. 20)

## Exemple de conflit shift/reduce: 1+2\*3



### > Solution n°44 (exercice p. 20)

## Exercice tiré de "Lex & Yacc" (O'Reilly), pp 58-66.

### Spécification Yacc

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

%token NB
%left '-' '+'
%left '*' '/'
%nonassoc MOINS

%%

S : E
    {printf("résultat : %d\n", $1);};

E :
E '+' E
| E '-' E
| E '*' E
| E '/' E
| '-' E %prec MOINS
| '(' E ')'
| NB
;
%{

int yyerror( char * s ) {
    fprintf( stderr, "%s\n", s );
    exit(1);
}

int main() {
    while (1)
        yyparse();
}
}
```

### Spécification Lex

```
%{
#include "y.tab.h"
%}
%%

[0-9]+ { yyval = atoi(yytext); return NB; }
[ \t]   {; }
\n      { return 0; }
.       { return yytext[0]; }

%%
```



Pour compiler le tout :

```
$ yacc -d fichier.y
$ lex fichier.l
$ gcc -o calculatrice y.tab.c lex.yy.c -ll
$ ./calculatrice
1+2
résultat : 3
```

### > Solution n°45 (exercice p. 20)

Le fait que les noms des variables sont de simples lettres nous permet de simplifier la table des symboles. En effet, cette lettre permet d'indexer la table des symboles (que l'on nomme "variables").

Spécification Yacc

```
%{
#include < stdio.h >
doublevariables[26];
%}

%union {
    double val;
    int varnum;
}

%token < varnum > ident
%token < val > NB
%token affect
%left '-' '+'
%left '*' '/'
%nonassoc MOINS

%type < val > E
%%
```

*L :*

```
I
| L ';' I
;
```

*I :*

```
ident affect E { variables[$1] = $3; }
| E           { printf( "résultat : %g\n", $1 ); }
;
```

```

E :
E +' E           { $$ = $1 + $3; }
| E '-' E       { $$ = $1 - $3; }
| E '*' E       { $$ = $1 * $3; }
| E '/' E       { $$ = $1 / $3; }
| '-' E %prec MOINS { $$ = -$2; }
| '(' E ')'    { $$ = $2; }
| NB            { $$ = $1; }
| ident          { $$ = variables[$1]; }
;
%%

int yyerror( char * s ) {
    fprintf( stderr, "%s\n", s );
    exit(1);
}

int main() {
    while (1)
        yyparse();
}

```

### Spécification Lex

```

%{
#include "y.tab.h"
%}
%%

[0-9]+(\.[0-9]+)? { yyval.val = atof(yytext); returnNB; }
[ \t]             {; }
" := "            {return affect; }
[a-z]             { yyval.varnum = yytext[0] - 'a'; return ident ; }
" + "
" - "
" * "
" / "
" _ "
" ( "
" ) "
";"              {return ';' ; }

.                  { printf("Erreur lexicale.\n"); }

%%

```

### > Solution n°46 (exercice p. 21)

Types d'implémentation : listes chainées, tableaux, table de hachages, arbres, etc.  
La plus utilisée: table de hachage.



> **Solution n°47** (exercice p. 21)

Avec une table de hachage:

```
typedef struct_symbole {
    char *nom;
    double valeur;
    struct _symbole *suivant;
} symbole;

#define TAILLE 103 /* nombre premier de préférence */

symbole *table[TAILLE];

int hash( char *nom ){
    int i, r;
    int taille = strlen(nom);
    r = 0;
    for ( i = 0; i < taille; i ++ )
        r = ((r << 8) + nom[i]) % TAILLE;
    return r;
}

void table_reset() {
    int i;
    for ( i = 0; i < TAILLE; i ++ )
        table[i] = NULL;
}
```

```
symbole * inserer( char *nom ) {
    int h;
    symbole *s;
    symbole *precedent;
    h = hash(nom);
    s = table[h];
    precedent = NULL;
    while ( s != NULL ) {
        if ( strcmp( s->nom, nom ) == 0 )
            return s;
        precedent = s;
        s = s->suivant;
    }
    if ( precedent == NULL ){
        table[h] = (symbole *) malloc(sizeof(symbole));
        s = table[h];
    }
    else {
        precedent->suivant = (symbole *) malloc(sizeof(symbole));
        s = precedent->suivant;
    }
    s->nom = strdup(nom);
    s->suivant = NULL;
    return s;
}
```

### > Solution n°48 (exercice p. 22)

#### exo1.h

```
#ifndef _EXO1_H
#define _EXO1_H

typedef struct _symbole {
    char *nom;
    double valeur;
    struct _symbole *suivant;
} symbole;

void table_reset();
symbole *inserer( char *nom );
```

```
#endif
```

#### exo1.l



```
%{
#include <stdlib.h>
#include <string.h>
#include "exo1.h"
#include "y.tab.h"
%}
%%
[0-9]+(\.[0-9]+)?      { yyval.val = atof(yytext); returnNB; }
[ \t\n]                  /* rien faire */
[a-zA-Z][a-zA-Z]*        { yyval.var = inserer(yytext); return NOM; }
"+"
"-"
"*"
"/"
"-"
 "("
 ")"
";"
.

{ printf("Erreur lexicale.\n"); }

#define TAILLE 103
symbole *table[TAILLE];

int hash( char *nom ) {
    int i, r;
    int taille = strlen(nom);
    r = 0;
    for ( i = 0; i < taille; i++ )
        r = ((r << 8) + nom[i]) % TAILLE;
    return r;
}

void table_reset() {
    int i;
    for ( i = 0; i < TAILLE; i++ )
        table[i] = NULL;
}
```

```
symbole * inserer( char *nom ) {
    int h;
    symbole *s;
    symbole *precedent;
    h = hash(nom);
    s = table[h];
    precedent = NULL;
    while ( s != NULL ) {
        if ( strcmp( s->nom, nom ) == 0 )
            return s;
        precedent = s;
        s = s->suivant;
    }
    if ( precedent == NULL ){
        table[h] = (symbole *) malloc(sizeof(symbole));
        s = table[h];
    }
    else {
        precedent->suivant = (symbole *) malloc(sizeof(symbole));
        s = precedent->suivant;
    }
    s->nom = strdup(nom);
    s->suivant = NULL;
    return s;
}
```

exo1.y : Attention, j'ai simplifié le symbole d'affectation de ":=" en le remplaçant par un '=' : ainsi, l'opérateur d'affectation devient un simple caractère retourné par lex

```
%{
#include <stdio.h>
#include "exo1.h"
%}
```

```
%union {
    double val;
    symbole *var;
}
```

```
%token < var > NOM
%token < val > NB
%left '-' '+'
%left '*' '/'
%nonassoc MOINS
```

```
%type < val > E
%%
```

*L* :

```
I
| L ';' I
;
```

*I* :

<i>NOM' ='</i> E           E         ;	{ [\$1 -> valeur = \$3; }         { printf( "résultat : %g\n", \$1 ); }
--	---

*E* :

<i>E' +' E</i>   <i>E' '-' E</i>   <i>E' '*' E</i>   <i>E' '/' E</i>   ' <i>'-' E</i> %prec MOINS           ' <i>(' E ')'</i>   NB           NOM         ;	{ \$\$ = \$1 + \$3; }         { \$\$ = \$1 - \$3; }         { \$\$ = \$1 * \$3; }         { \$\$ = \$1 / \$3; }         { \$\$ = -\$2; }         { \$\$ = \$2; }         { \$\$ = \$1; }         { \$\$ = \$1 -> valeur; }
--	--

```
int yyerror( char * s ) {
    fprintf( stderr, "%s\n", s );
    exit(1);
}

int main() {
    while (1)
        yyparse();
}
```

> **Solution n°49** (exercice p. 22)

exo2\_1.h

```
/* structures de données */

typedef enum { ENTIER, CHAINE } type_t;

typedef struct _param_t {
    type_t type;
    char *nom;
} param_t;

typedef struct _liste_t {
    param_t param;
    struct _liste_t *suivant;
} liste_t;
```

exo2\_1.l

```
%{

#include < string.h >
#include "exo2_1.h"
#include "y.tab.h"

%}

%%

[ \t\n]           { /* supprimer les blancs */ }
int               { return int; }
string            { return string; }
[_a-zA-Z][_a-zA-Z0-9]* { yyval.nom = strdup(yytext); return ident; }
"("              { return '('; }
")"              { return ')'; }
","              { return ','; }
";"              { return ';' ; }
.
{ printf ("Erreur lexicale. \n"); }

%%
```

## exo2\_1.y

```
%{

#include < stdio.h >
#include < stdlib.h >
#include < assert.h >
#include "exo2_1.h"

liste_t * creer_liste( param_t p ); /* crée une nouvelle liste en l'initialisant avec un élément de type param_t */
liste_t * concatener_listes(liste_t *l1, liste_t *l2 ); /* crée une nouvelle liste en concaténant deux sous listes */
void afficher_liste( liste_t *liste ); /* parcourt et affiche le contenu d'une liste */
void yyerror(char *s);

}

%union {
    char *nom;
    type_t type;
    param_t param;
    liste_t *liste;
}

%token int string
%token < nom > ident

%type < type > T
%type < param > P
%type < liste > L
```

*%%*

```

/* on rajoute la règle suivante pour pouvoir parser plusieurs déclarations : */
S :
  D
|  S D
;

D :
  T ident'(' L ')' ';' {
    printf( "Fonction : %s\n", $2 );
    printf( "Type : %s\n", ($1 == ENTIER)? "int" : "string" );
    printf( "Arguments : " );
    afficher_liste($4);
    printf("\n\n");
  }
|  T ident'(' ')' ';' {
    printf( "Fonction : %s\n", $2 );
    printf( "Type : %s\n", ($1 == ENTIER)? "int" : "string" );
    printf( "Pas d'arguments.\n\n" );
  }
;

L :
  P      { $$ = creer_liste($1); }
|  L ',' P  { $$ = concatener_listes( $1, creer_liste($3)); }
;

P :
  T ident { $$ .type = $1; $$ .nom = $2; }
;

T :
  int { $$ = ENTIER; }
|  string { $$ = CHAINE; }
;

```

*%%*

```

liste_t *creer_liste( param_t p ) {
    liste_t *liste;
    liste = (liste_t *) malloc(sizeof(liste_t));
    assert( liste != NULL );
    liste->param = p;
    liste->suivant = NULL;
    return liste;
}

liste_t *concatener_listes( liste_t *l1, liste_t *l2 ) {
    liste_t *l = l1;
    if ( l == NULL )
        return l2;
    while ( l->suivant != NULL )
        l = l->suivant;
    l->suivant = l2;
    return l1;
}

void afficher_liste( liste_t *liste ) {
    liste_t *l;
    for ( l = liste; l != NULL; l = l->suivant ) {
        if ( l != liste )
            printf(",");
        printf(" %s (%s)", l->param.nom,
               (l->param.type == ENTIER)?"int" : "string");
    }
}

void yyerror( char *s ) {
    fprintf( stderr, "%s\n", s );
}

int main() {
    return yyparse();
}

```

*Pour compiler et tester :*

```

$ yacc -d exo2_1.y
$ lex exo2_1.l
$ gcc -g -o exo2_1 lex.yy.c y.tab.c -ll
$ ./exo2_1 < test_exo2.txt

```

### > Solution n°50 (exercice p. 22)

**exo2\_2.y** : il faut créer une table des symboles pour les fonctions

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "exo2_1.h"

liste_t * creer_liste( param_t p );
liste_t* concatener_listes(liste_t *l1, liste_t *l2 );

void yyerror(char *s );

/* on rajoute une structure de données contenant
les informations rattachées pour les fonctions */

typedef struct _fonction_t {
    type_t type;
    char *nom;
    liste_t *arguments;
    struct _fonction_t *suivant;
} fonction_t;

/* et une table de symboles ( similaire à celle de l'exo 1 ) : */
fonction_t *ajouter_fonction(type_t type, char *nom, liste_t *args );
void afficher_fonction( fonction_t *fonction );

%}

%union {
    char *nom;
    type_t type;
    param_t param;
    liste_t *liste;
}

%token int string
%token <nom> ident

%type <type> T
%type <param> P
%type <liste> L
```



*%%*

*S :*

| *D*  
| *S D*  
| ;

*D :*

*T ident ' ( L ') ' ; {*  
*fonction\_t \*fonction = ajouter\_fonction( \$1, \$2, \$4 );*  
*printf( "Type : %s\n", ( \$1 == ENTIER )? "int" : "string" );*  
*if ( fonction != NULL )*  
*afficher\_fonction(fonction);*  
}  
| *T ident' ( ) ' ; {*  
*fonction\_t \*fonction = ajouter\_fonction( \$1, \$2, NULL );*  
*if ( fonction != NULL )*  
*afficher\_fonction(fonction);*  
}  
; ;

*L :*

*P { \$\$ = creer\_liste(\$1); }*  
| *L ', ' P { \$\$ = concatener\_listes( \$1, creer\_liste(\$3)); }*  
; ;

*P :*

*T ident { \$\$ .type = \$1; \$\$ .nom = \$2; }*  
; ;

*T :*

*int { \$\$ = ENTIER; }*  
| *string { \$\$ = CHAINE; }*  
; ;

*%%*

```

liste_t *creer_liste( param_t p ) {
    liste_t *liste;
    liste = (liste_t *) malloc(sizeof(liste_t));
    assert( liste != NULL );
    liste->param = p;
    liste->suivant = NULL;
    return liste;
}

liste_t *concatener_listes( liste_t *l1, liste_t *l2 ) {
    liste_t *l = l1;
    if ( l == NULL )
        return l2;
    while ( l->suivant != NULL )
        l = l->suivant;
    l->suivant = l2;
    return l1;
}

void afficher_liste( liste_t *liste ) {
    liste_t *l;
    for ( l = liste; l != NULL; l = l->suivant ) {
        if ( l != liste )
            printf(",");
        printf(" %s (%s)", l->param.nom,
               (l->param.type == ENTIER)?"int" :"string");
    }
}

int listes_egales( liste_t *l1, liste_t *l2 ) {
    liste_t *liste;
    for( liste = l1; liste != NULL; liste = liste->suivant ) {
        if ((l2 == NULL)|| (l2->param.type != liste->param.type))
            return 0;
        l2 = l2->suivant;
    }
    if ( l2 != NULL )
        return 0;
    return 1;
}

```



```
/* gestion de la table de symboles */  
  
#define TAILLE 103  
  
fonction_t *table[TAILLE];  
  
int hash( char *nom ) {  
    int i, r;  
    int taille = strlen(nom);  
    r = 0;  
    for ( i = 0; i < taille; i ++ )  
        r = (( r << 8 ) + nom[i]) % TAILLE;  
    return r;  
}  
  
void init_table() {  
    int i;  
    for( i = 0; i < TAILLE; i ++ )  
        table[i] = NULL;  
}
```

```

/* ajouter_fonction(..) renvoie NULL si la fonction a déjà été déclarée */

fonction_t * ajouter_fonction( type_t type, char *nom, liste_t *args ) {
    int h;
    fonction_t *f;
    fonction_t *precedent;
    fonction_t *nouvelle_fonction;
    h = hash(nom);
    f = table[h];
    precedent = NULL;
    while ( f != NULL ) {
        if ( strcmp( f->nom, nom ) == 0 ) {
            /* on a trouvé une fonction portant le même nom */
            if (( f->type == type )&&( listes_egales( f->arguments, args)))
                printf( "Re-déclaration cohérente de la fonction %s\n", f->nom );
            else
                printf( "Re-déclaration incohérente de la fonction %s\n", f->nom );
            return NULL;
        }
        precedent = f;
        f = f->suivant;
    }
    nouvelle_fonction = (fonction_t *) malloc(sizeof(fonction_t));
    assert(nouvelle_fonction != NULL);
    if ( precedent == NULL ) {
        table[h] = nouvelle_fonction;
        f = table[h];
    }
    else {
        precedent->suivant = nouvelle_fonction;
        f = precedent->suivant;
    }
    f->type = type;
    f->nom = strdup(nom);
    f->arguments = args;
    f->suivant = NULL;
    return f;
}

```

```

void afficher_fonction( fonction_t *fonction ) {
    printf( "Fonction : %s\n", fonction->nom );
    printf( "Type : %s\n", ( fonction->type == ENTIER)? "int" : "string" );
    printf( "Arguments : " );
    if ( fonction->arguments != NULL )
        afficher_liste( fonction->arguments );
    else
        printf( "aucun" );
    printf( "\n\n" );
}

void yyerror( char *s ) {
    fprintf( stderr, "%s\n", s );
}

int main() {
    return yyparse();
}

```

*Pour compiler et tester :*

```

$ yacc -d exo2_2.y
$ lex exo2_1.l
$ gcc -g -o exo2_2 lex.yy.c y.tab.c -llex
$ ./exo2_2 < test_exo2.txt

```

### > Solution n°51 (exercice p. 23)

Une interface de table de symboles possible:

```

typedef enum { INT, STRING } type_t;
typedef struct _symbole_t {
    char *nom;
    type_t type;
    int taille;
    int position;
    struct _symbole_t *suivant;
} symbole_t;

symbole_t * ajouter( char * nom );
symbole_t * rechercher( char * nom );

```

### > Solution n°52 (exercice p. 24)

Actions sémantiques:

Variable globale : `int pos = 0;`

```
%union {
    char *name;
    type_t type;
    int entier;
}

%type < entier > Q
%type < type > T
%token int string

%token < name > ident
%token < entier > const
```

Attention, j'écris ici en pseudo code !

```
L -> D | L D      {}
T -> int           {$$ = INT; }
T -> string        {$$ = STRING; }
Q -> [ const ]     { $$ = $2; }
Q -> epsilon       { $$ = 1; } // un seul element si non tableau

D -> T Q ident   {
    symbole_t *s = rechercher($3);
    if ( s != NULL )
        erreur( "Variable déjà déclarée !");
    else {
        s = ajouter($3);
        s->type = $1;
        s->taille = taille_type($1) * $2;
        s->position = pos;
        pos += s->taille;
    }
}
```

### > Solution n°53 (exercice p. 24)

Oui, mais cela serait compliqué à cause des déclarations multiples avec le même nom dans des blocs imbriqués. Il faudrait inclure de nouvelles informations pour chaque symbole. Mécanisme plus adapté : une table de symboles par bloc. Par exemple une pile de tables de symboles.



*Nouvelle interface :*

```
symbole_t * ajouter( table_t table, char * nom );
symbole_t * rechercher( table_t table, char * nom );
void supprimer( table_t table, char * nom );
```

*On a aussi besoin d'un mécanisme de pile :*

```
objet top();
void push(objet o);
objet pop();
```

*La règle  $I \rightarrow \{ L I \} | \{ I \} | S$  est inadaptée, on change légèrement la grammaire :*

$I \rightarrow$	<i>entrée <math>L I</math> sortie</i>
$I \rightarrow$	<i>entrée <math>I</math> sortie</i>
$I \rightarrow$	<i>S</i>
<i>entrée</i> $\rightarrow$	{ { <i>table</i> = nouvelle_table(); push( <i>table</i> ); }
<i>sortie</i> $\rightarrow$	{ { <i>table</i> = pop(); détruire_table( <i>table</i> ); }

Les autres actions sémantiques sont identiques à celles de l'exercice précédent sauf que l'on utilise la table courante top().

## > Solution n°54 (exercice p. 24)

Je suppose que les tables sont chaînées afin de pouvoir les accéder les unes après les autres.

On change une action sémantique:

```
D → T Q ident {
    symbole_t *s = NULL;
    for (table = top(); table != NULL; table = table-> suivant) {
        s = rechercher( table, $3 );
        if (s != NULL)
            break;
    }
    if (s != NULL) {
        if (table == top())
            erreur( "Variable déjà déclarée !" );
        else {
            avertissement( "La variable %s en masque une autre.", $3 );
            s = NULL;
        }
        if (s == NULL) {
            s = ajouter( top(), $3 );
            s-> type = $1;
            s-> taille = taille_type($1) * $2;
            s-> position = pos;
            pos += s-> taille;
        }
    }
}
```

## > Solution n°55 (exercice p. 25)

Du code assembleur n'est autre que du texte brut ayant une certaine structure.

Dans quoi allons nous stocker le code que l'on génère ?

Des solutions évidentes et simples : listes chaînées, tableaux, etc.

D'autres structures de données sophistiquées existent : graphe de flot de contrôle, etc.

En général, les compilateurs passent d'abord par une représentation intermédiaire (un langage plus bas niveau, mais indépendant de la machine cible) avant de générer un code assembleur dépendant de la machine cible.

Dans ce TD, nous court-circuitons la représentation intermédiaire. En d'autres termes, nous générerons directement du code assembleur.

## > Solution n°56 (exercice p. 25)

Les variables scalaires peuvent être allouées avec plusieurs méthodes (au choix) :

- sur la pile : variables locales. Une variable est substituée par un calcul d'adresse %ebp(offset)
- dans la mémoire (mais en dehors de la pile) : variables globales, externes, etc. La variable devient une étiquette.
- sur la tas (utilisant malloc). La variable devient un pointeur
- dans des registres : pour peu de variables uniquement. Une variable devient un nom de registre processeur.

## > Solution n°57 (exercice p. 26)

L'utilisation de la pile comme moyen d'évaluer une expression arithmétique suggère que cette expression est écrite en notation post-fixée. Par exemple,  $a * b + c$  serait écrite  $ab * c+$ . Ceci se traduirait par :

*empiler a*

*empile b*

*dépiler a, dépiler b, effectuer a \* b, empiler le résultat*

*empiler c*

*dépiler c, dépiler a \* b, faire l'addition, et empiler le résultat*

Or, cela tombe bien, car une analyse LR avec yacc parcourt l'expression (la grammaire) dans cette ordre post-fixé. Ceci ne serait pas aussi aisés avec une analyse LL (descendante).

## Hypothèses

On suppose que chaque non-terminal a un attribut, que nous appelons "code", qui est une liste linéaire chaînée. Cette liste contient le code que l'on génère au fur et à mesure (des chaînes de caractères). Ainsi, j'utilise une notation orientée objet comme ceci :

`code.insert("add x,y,z")` : insérer le texte assembleur "add x,y,z" dans la liste (code).

`code.init()` : initialise la chaîne à vide.

`code.init("add x,y,z")` : initialisation avec une chaîne donnée en paramètre.

On suppose que les fonctions insert et init admettent un nombre quelconque de paramètres. De plus, nous supposons que le token "ident" contient un attribut "nom". Pour simplifier notre exercice, nous supposons que les variables sont

allouées en mémoire. Ainsi, on peut y accéder par leur nom (étiquette) en code assembleur. Si ces variables étaient allouées sur la pile, il faudrait accéder à la table des symboles pour extraire le déplacement (offset) de la variable par rapport à la base de la pile %ebp. Mais, supposons que les variables sont en mémoire pour simplifier !

Nous supposons également que le token "const" a un attribut numérique appelé "valeur".

On suppose que l'on a déclaré dans yacc une chaîne de caractère temporaire "temp" utilisée par les actions sémantiques pour générer une instruction temporaire courante.

*Voici comment effectuer des calculs en x386.*

```
// %eax + %ecx -> %eax :
    addl %ecx,%eax

// %eax - %ecx -> %eax :
    subl %ecx,%eax

// %eax * %ecx -> %eax :
    imull %ecx

// %eax / %ecx -> %eax
    cltd
    idivl %ecx
```

Dans ce qui suit, le symbole d'affectation := est remplacé par le token "affect".

Actions sémantiques le code final est dans la liste S.code

S ->

```
I {$$.code.insert($1.code);}

; S
| {$$.code.init();} /* initialiser la liste d'instructions à liste vide */
```

*Traduction de I –> ident affect E*

<pre>&lt; E.code &gt; popl %eax movl %eax ident.nom pushl %eax pushl \$chaine call printf addl \$4 %%esp</pre>	<i>// on laisse %eax sur le sommet de pile (par convention)</i>
--	---

*I –> ident affect E*

{

  \$\$.code.init(\$3.code);

*/\* par convention, le résultat de l'expression est dans le sommet de la pile \*/*  
*/\* donc, générer le code qui dépile le résultat et l'affecte à la variable stoquée en mémoire. Le nom de la variable est dans l'attribut ident.nom \*/*

  sprintf(temp, "movl %%eax %%s", \$1.nom); */\* générer l'instruction \*/*  
  \$\$.code.insert("popl %%eax", temp); */\* puis l'injecter dans le code \*/*

*/\* Maintenant, on peut générer le printf("%d", résultat) \*/*  
*/\* Le passage des paramètres se fait par adresse,*  
*dans l'ordre inverse de leur apparition dans la fonction. \*/*  
*/\* Il faut d'abord empiler la variable ensuite l'adresse de la chaîne. \*/*

  \$\$.code.insert("pushl %%eax", "pushl \$chaine", "call printf", "addl \$4, %%esp");  
*/\* On ne dépile que 4 octets afin de laisser le résultat de E au sommet de la pile \*/*  
*/\* chaine = "%d" \*/*  
*/\* après l'appel de printf, on doit dépiler les paramètres (par convention) \*/*  
}

*Traduction de I –> E*

<pre>&lt; E.code &gt; pushl \$chaine call printf addl \$4 %%esp</pre>	<i>// le résultat est toujours sur la pile</i>
---	--

*I –> E*

  {  
*/\* par convention, le résultat de l'expression est dans le sommet de la pile \*/*  
*/\* donc, afin de l'afficher avec printf, il suffit d'empiler l'adresse de la chaîne et faire un call \*/*  
  \$\$.code.init(\$1.code, "pushl \$chaine", "call printf", "addl \$4, %%esp");  
*/\* On ne dépile que 4 octets afin de laisser le résultat de E au sommet de la pile \*/*  
}



Traduction de  $E \rightarrow E1 + T$

< E1.code >
< T.code >
popl %%ecx
popl %%eax
addl %%ecx, %%eax
pushl %%eax

$E \rightarrow E + T$

{

/\* si on a  $a + b$ , on génère d'abord le code qui calcule  $a$ , ensuite le code qui calcule  $b$ ,  
on dépile  $b$  et  $a$ , on fait l'addition, et on empile le résultat \*/  
/\* le code qui calcule  $a$  est inséré le premier car on doit avoir le résultat de  $b$  au sommet de la pile d'après l'énoncé du TD \*/

```
 $$ .code.init($1.code,$3.code, "popl %%ecx", "popl %%eax", "addl %%ecx, %%eax", "pushl %%eax" );
 }
```

Traduction de  $E \rightarrow E1 - T$

< E1.code >
< T.code >
popl %%ecx
popl %%eax
subl %%ecx, %%eax
pushl %%eax

$E \rightarrow E - T$

{

/\* idem \*/  
\$\$ .code.init(\$1.code,\$3.code, "popl %%ecx", "popl %%eax", "subl %%ecx, %%eax", "pushl %%eax" );

}

$E \rightarrow T$

{

```
 $$ .code.init( $1.code );
 }
```

Traduction de  $T \rightarrow T1 * F$

< T1.code >
< F.code >
popl %%ecx
popl %%eax
imul %%ecx
pushl %%eax

$T \rightarrow T * F$

{

```
 $$ .code.init($1.code,$3.code, "popl %%ecx", "popl %%eax", "imul %%ecx", "pushl %%eax" );
 /* Ici, la multiplication donne un résultat dans deux registres edx : eax. */
 /* Nous devrions normalement empiler le résultat sous forme de deux mots edx : eax */
 /* et se débrouiller pour que le résultat de l'expression finale soit remis en deux mots aussi. */
 /* Mais bon, on en est pas à ce détail près ! */
 }
```

Traduction de T -> T1 / F

```
< T1.code >
< F.code >
popl %%ecx
popl %%eax
cltd
idvl %%ecx
pushl %%eax
```

T -> T / F

```
{
$$code.init($1.code,$3.code, "popl %%ecx", "popl %%eax", "cltd", "idvl %%ecx", "pushl %%eax");
/* cltd sert à étendre l'opérande de la division sur deux registres : EAX -> EDX : EAX */
}
```

T -> F

```
{
$$code.init($1.code );
}
```

F -> ident

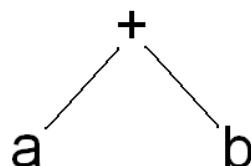
```
{
/* on empile la variable */
sprintf( temp, "pushl %s", $1.nom );
$$code.init( temp );
}
```

F -> const

```
{
/* on empile la constante numérique */
sprintf( temp, "pushl $%d", $1.valeur );
$$code.init( temp );
}
```

## > Solution n°58 (exercice p. 26)

Pour comprendre, considérons l'arbre abstrait d'une expression arithmétique :



Par convention :

- on stocke les résultats des nœuds intermédiaires dans le registre %eax. Ainsi, le résultat de l'addition ci-dessous est mis dans %eax.
- on génère le code de B, le fils droit. Le résultat est dans %eax par convention, puis on l'empile.
- on génère le code de A, le fils gauche. Le résultat est dans %eax par convention. Dépiler dans %ecx pour récupérer le résultat de B, effectuer le calcul et stocker dans %eax.

Le parcours LR suit ce schéma là. Il nous reste à écrire les actions sémantiques.

Actions sémantiques : le code final est dans la liste S.code

$S \rightarrow$  $I \{ $$ .code.insert( \$1.code ); \}$  $; S$  $| \{ $$ .code.init(); \} /* initialiser la liste d'instructions à liste vide */$ *Traduction de I → ident affect E*

< E.code >
<code>movl %eax ident.nom</code>
<code>pushl %eax</code>
<code>pushl \$chaine</code>
<code>call printf</code>
<code>addl \$4 %%esp</code>
<code>popl %eax</code>
// enlever \$chaine de la pile
// restaurer %eax

 $I \rightarrow ident affect E$ 

{

 $\{ $$ .code.init( \$3.code ); /* insérer le code calculant l'expression */$  $/* par convention, le résultat de l'expression est dans %eax */$  $/* donc, générer le code qui stocke %eax dans la variable ident */$  $sprintf( temp, "movl %%eax, %s", \$1.nom ); /* générer l'instruction */$  $$$ .code.insert( temp ); /* puis l'injecter dans le code */$  $/* Maintenant, on peut générer le printf("%d", résultat) */$  $/* Le passage des paramètres se fait par adresse,$  $dans l'ordre inverse de leur apparition dans la fonction. */$  $/* Il faut d'abord empiler le résultat ensuite l'adresse de la chaine. */$  $$$ .code.insert( "pushl %%eax", "pushl $chaine", "call printf", "addl $4, %%esp", "popl %%eax" );$ 

}

*Traduction de I → E*

< E.code >
<code>pushl %eax</code>
<code>pushl \$chaine</code>
<code>call printf</code>
<code>addl \$4 %%esp</code>
<code>popl %eax</code>
// restaurer %eax

 $I \rightarrow E$ 

{

 $/* par convention, le résultat de l'expression est dans %eax */$  $/* Empiler %eax ensuite la chaine avant de faire le call de printf. */$  $$$ .code.init( \$1.code, "pushl %%eax", "pushl $chaine", "call printf", "addl $4, %%esp", "popl %%eax" );$ 

}

*Traduction de E -> E1 + T*

< T.code >
pushl %eax
< E1.code >
popl %ecx
addl %ecx %eax

*E -> E + T*

```
{
$$.code.init( $3.code, "pushl %%eax", $1.code, "popl %%ecx", "addl %ecx, %%eax");
}
```

*Traduction de E -> E1 - T*

< T.code >
pushl %eax
< E1.code >
popl %ecx
subl %ecx %eax

*E -> E - T*

```
{
/* idem */
$$.code.init( $3.code, "pushl %%eax", $1.code, "popl %%ecx", "subl %ecx, %%eax");
}
```

*E -> T*

```
{
/* idem */
$$.code.init( $1.code );
}
```

*Traduction de T -> T1 \* F*

< F.code >
pushl %eax
< T1.code >
popl %ecx
imul %ecx

*T -> T \* F*

```
{
$$.code.init( $3.code, "pushl %%eax", $1.code, "popl %%ecx", "imul %ecx");
}
```

*Traduction de T -> T1 / F*

< F.code >
pushl %eax
< T1.code >
popl %%ecx
cltd
idivl %%ecx

*T -> T / F*

```
{
  $$ .code.init( $3 .code, " pushl %%eax ", $1 .code, " popl %%ecx ", " cltd ", " idivl %%ecx ");
}
```

*T -> F*

```
{
  $$ .code.init( $1 .code );
}
```

*F -> ident*

```
{
/* On charge la variable dans %eax */
sprintf(temp, " movl %s, %%eax ", $1 .nom);
$$ .code.init( temp );
}
```

*F -> const*

```
{
/* On met la constante dans %eax */
sprintf(temp, " movl %d, %%eax ", $1 .valeur);
$$ .code.init( temp );
}
```

### > Solution n°59 (exercice p. 26)

Le code généré n'est pas exécutable. Il faut l'assembler puis le linker.

Mais tel qu'il est, le programme assembleur n'est pas complet. Il manque le prologue et l'épilogue comme ceci:

```
.global main
.type main, @function
main :
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp /* allocation de variables locales */

    ...
    leave
    ret

.chaine : .string "%d" /* déclaration des chaînes de caractères */
.comm toto ,4,4 /* variables globales */
```

## > Solution n°60 (exercice p. 26)

Le passage par valeur implique de copier la chaîne sur la pile avant d'appeler printf. Pour copier des données d'une zone mémoire à une autre, on a le choix entre écrire une boucle qui copie les caractères un à un, ou bien d'appeler la fonction strcpy de la libC, ou bien utiliser une instruction assembleur MOVS du i386.

Dans cette solution, je vous propose d'utiliser strcpy.

Le code généré pour l'appel de fonction strcpy(@src, @dest) doit d'abord empiler l'adresse destination (adresse sommet de pile), ensuite l'adresse de la chaîne source.

Exemple : je veux faire printf("toto", x);  
*pushl x*  
*/\* allouer une zone de 128 octets au sommet de la pile afin de copier la chaîne \*/*  
*subl \$128 %esp*  
*/\* maintenant je dois copier une chaîne de 128 octets sur la pile \*/*  
*/\* strcpy(%esp, \$chaine)*  
*pushl %esp*  
*pushl \$chaine*  
*call strcpy // elle copie les caractères dans le sens des adresses croissantes, contrairement à l'orientation de la pile*  
*addl \$8 %esp // faut-il vider la pile des deux précédents paramètres ? c'est les 2 pointeurs de 4 octets.*

*call printf /\* appel avec passage de paramètres par valeur \*/*  
*addl \$132 %esp (dépiler 128 + 4 octets : chaîne + le x initial)*

Evidemment, ce passage par valeur n'a aucune chance de fonctionner avec le printf standard puisqu'il ne prend en compte que des passages par valeur.

Etat de la pile : juste avant call strcpy

%esp ->	\$chaine
	%esp - "2"
	zone de 128 octets
	x

## > Solution n°61 (exercice p. 26)

La convention du langage C suggère que les paramètres s'empilent dans le sens inverse de leur apparition (de droite à gauche). Par exemple, l'appel de fonction toto(a,b,c,d) génère un code qui empile les paramètres dans l'ordre d c b a. Les paramètres peuvent être des expressions, donc il faut d'abord générer le code qui

évalue ces expressions.

Après avoir empilé les paramètres, puis avoir généré le call, il ne faut pas oublier de vider la pile.

*Exemple : foo(a + b, c + d) s'écrit*

<i>empiler</i>	<i>d</i>
<i>empiler</i>	<i>c</i>
<i>depiler</i>	<i>vers eax</i>
<i>depiler</i>	<i>vers ecx</i>
<i>eax + ecx</i>	$\rightarrow eax$
<i>empiler eax</i> : nous avons évalué le paramètre <i>c + d</i> qui se trouve empilé maintenant	

<i>empiler</i>	<i>b</i>
<i>empiler</i>	<i>a</i>
<i>depiler</i>	<i>vers eax</i>
<i>depiler</i>	<i>vers ecs</i>
<i>eax + ecx</i>	$\rightarrow eax$
<i>empiler eax</i> : nous avons évalué le paramètre <i>a + b</i> qui se trouve empilé maintenant	

<i>call</i>	<i>foo</i>
<i>addl</i>	<i>\$8, %esp &lt;- vider la pile des 2 paramètres précédemment empilés.</i>

*pushl %eax <- le résultat de la fonction est dans %eax, empilé par convention*  
Afin de connaître le nombre d'éléments empilés, les actions sémantiques doivent maintenir un compteur.

Soit L.NbArgs un attribut contenant le nombre de paramètres dans la liste L.

<i>Traduction de F -&gt; ident(E1, ..., Ek)</i>
<i>&lt; Ek.code &gt;</i>
<i>&lt; Ek - 1.code &gt;</i>
<i>...</i>
<i>&lt; E1.code &gt;</i>
<i>call foo</i>
<i>addl 4 * k, %esp</i>
<i>pushl %eas</i>
<i>// le résultat de la fonction (dans %eax) est empilé</i>

*Attributs :* *L.NbArgs = nombre d'arguments dans la liste des paramètres. Ceci afin de générer l'instruction "addl 4 \* k %esp".*  
*L.Code = code évaluant les paramètres de la liste. Cet attribut contient les codes (Ek.code, Ek - 1.code, ..., E1.code)*

```

 $F \rightarrow ident(L)$ 
{
  $$.code.init();
  $$.code.insert($$.code);

  sprintf(inst,mp, "call %s", $1.nom); /* créer l'instruction du call... */
  $$.code.insert(inst,mp); /* ... et l'injecter dans le code */

  sprintf(inst,mp, "addl $%d %%esp", sizeof(int) * $3.NbArgs); /* créer l'instruction qui vide la pile d'exécution */
  $$.code.insert(inst,mp); /* ... et l'injecter dans le code */
  $$.code.insert("pushl %%eax");
}

 $L \rightarrow epsilon$ 
{
  $$.Args.init();
  $$.NbArgs = 0;
}

 $L \rightarrow E$ 
{
  $$.code.init($1.code); /* mettre le code qui évalue E dans la liste L.code */
  $$.NbArgs = 1; /* un seul paramètre jusqu'à présent. */
}

 $L \rightarrow L, E$ 
{
  $$.code = concatener($3.code, $1.code); /* attention, afin d'avoir une liste des paramètres triés de droite à gauche,
                                             on insère ici le code de E à gauche de la liste $1.code */
  $$.NbArgs = $1.NbArgs + 1; /* incrémenter le compteur des paramètres car nous avons un nouveau paramètre */
}

```

## > Solution n°62 (exercice p. 27)

### machine à registres

*Traduction de  $F \rightarrow ident(E_1, \dots, E_k)$*

$< E_k.code >$	
<i>pushl %%eax</i>	
$< E_{k-1}.code >$	
<i>pushl %%eax</i>	
<i>...</i>	
$< E_1.code >$	
<i>pushl %%eax</i>	
<i>call foo</i>	
<i>addl 4 * k, %%esp</i>	
<i>pushl %%eas</i>	
<i>// le résultat de la fonction (dans %%eax) est empilé</i>	



```

F -> ident(L)
{
  $$.code.init();
  $$.code.insert($3.code);

  sprintf(inst,mp, "call %s", $1.nom); /* créer l'instruction du call... */
  $$.code.insert(inst,mp); /* ... et l'injecter dans le code */

  sprintf(inst,mp, "addl $%d %%esp", sizeof(int) * $3.NbArgs); /* créer l'instruction qui vide la pile d'exécution */
  $$.code.insert(inst,mp); /* ... et l'injecter dans le code */
  // le résultat final de la fonction est laissé dans %eax
}

L -> epsilon
{
  $$.Args.init();
  $$.NbArgs = 0;
}

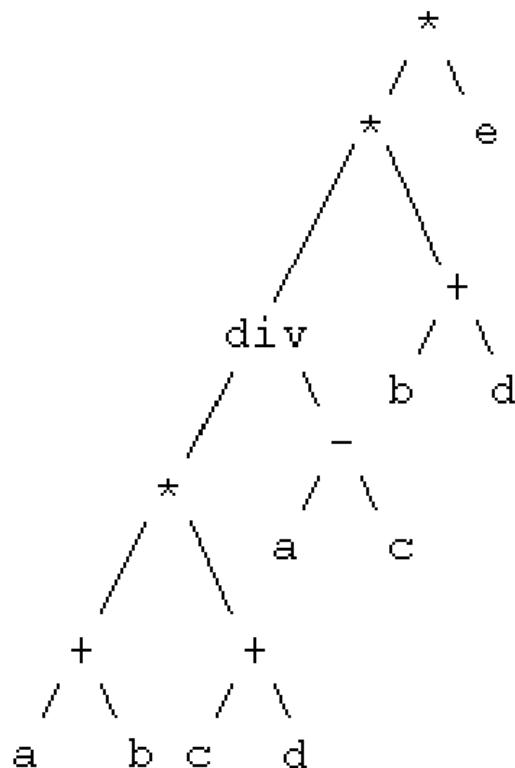
L -> E
{
  $$.code.init($1.code); /* mettre le code qui évalue E dans la liste L.code */
  $$.code.insert("pushl %%eax"); // ici, par rapport à la question 1.2 on ajoute "pushl %%eax"
  $$.NbArgs = 1; /* un seul paramètre jusqu'à présent. */
}

L -> L, E
{
  $$.code = concatener($3.code, "pushl %%eax", $1.code); // ici, par rapport à la question 1.2 on ajoute "pushl %%eax"
  $$.NbArgs = $1.NbArgs + 1;
}

```

### > Solution n°63 (exercice p. 29)

#### arbre d'expression binaire



## Algorithme : Sethi and Ullman 1970. Algorithme prouvé optimal.

### Première variante: calculer le nombre minimal de registres nécessaires pour l'évaluation d'un arbre abstrait.

L'algorithme calcule une étiquette/nombre  $L(u)$  pour chaque nœud  $u$  : cette étiquette correspond au nombre minimal de registres requis pour calculer l'expression dont la racine est  $u$ . Le code pour un sous-arbre  $u$  est généré de telle sorte à ce que le sous-arbre fils qui a la plus petite étiquette soit calculé le premier. Si les deux sous arbres fils ont la même étiquette, alors cela veut dire qu'il nous faut un registre supplémentaire.

La consommation en registres à chaque instruction dans le code généré a la propriété suivante :

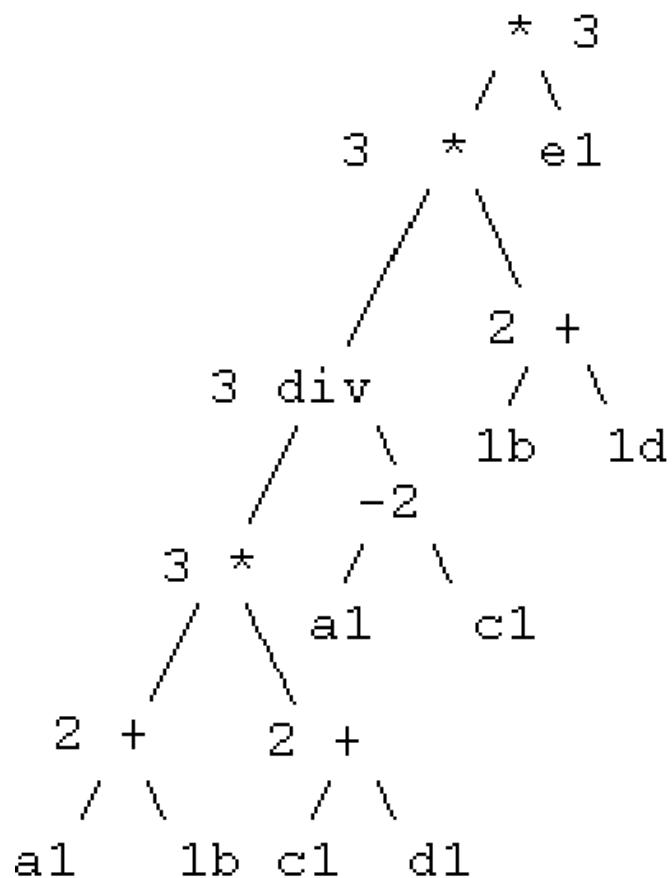
- Initialement, le besoin en registres est 0.
- En avançant dans le code généré instruction par instruction, si l'instruction est un load, alors le besoin de registres est incrémenté de 1.

Si c'est une opération arithmétique, la consommation en registres est décrémentée de 1 (puisque'on a libère deux registres opérandes, et on consomme un registre pour le résultat).

### Algorithme de Sethi Ullman pour la minimisation du nombre de registres requis : Cas sans spill:

```
Sethi_Ullman(u) /* u est la racine de l'arbre d'une expression arithmétique. u.op est l'opération arithmétique de cette racine. */  
{  
    si u est une feuille alors  
    {  
        L(u) <= 1; /* la feuille initiale correspond à un load */  
        générer le code du load;  
        /* si l'architecture peut executer des opérations mémoire – registre, alors  
        initialiser l'étiquette du fils droit à 0, et celle du fils gauche à 1, ou inversement */  
    }  
    sinon{  
        soit u1 le fils gauche et u2 le fils droit.  
        si L(u1) = L(u2) alors  
        {  
            L(u) <= L(u1) + 1; /* un registre supplémentaire est requis  
            pour stocker le résultat intermédiaire  
            d'un des fils */  
            Sethi_Ullman(u1); // générer récursivement le code de u1  
            Sethi_Ullman(u2); // puis le code de u2, ou l'inverse, c'est pareil */  
            générer le code de l'instruction "u1 u.op u2";  
        }  
        sinon  
        {  
            L(u) <= Max(L(u1), L(u2));  
            // soit u' le fils qui a la plus grande étiquette, u'' celui qui a la plus petite étiquette  
            Sethi_Ullman(u'); // générer récursivement le code du noeud qui a la plus grande étiquette L; /* son résultat est dans un registre */  
            Sethi_Ullman(u''); // générer ensuite récursivement le code du noeud qui a la plus petite étiquette L;  
            /* cet ordre d'évaluation garantit que l'on ne dépasse pas Max(L(u1), L(u2)) registres */  
            générer le code de l'instruction "u1 u.op u2";  
        }  
    }  
}
```

voici l'arbre étiqueté (il requiert 3 registres au minimum)



Voici l'ordre optimal d'évaluation de l'expression avec 3 registres (R1,R2,R3):

```

load  a, R1
load  b, R2
add   R1, R2, R1  /* R1 = a + b */

load  c, R2
load  d, R3
add   R2, R3, R2  /* R2 = c + d */

mult  R1, R2, R1  /* R1 = (a + b)(c + d) */

load  a, R2
load  c, R3
sub   R2, R3, R2  /* calcul de a - c */

div   R1, R2, R1  /* R1 = (a + b)(c + d)/(a - c) */

load  b, R2
load  d, R3
add   R2, R3, R2  /* R2 = b + d */

mult  R1, R2, R1  /* R1 = (a + b)(c + d)/(a - c)(b + d) */

load  e, R2
mult  R1, R2, R1  /* R1 = expression */

```

### > Solution n°64 (exercice p. 29)

#### nombre minimal de spill.

L'algorithme Sethi-Ullman précédent n'introduit pas de spill. Dans cette partie, nous donnons la deuxième variante de l'algorithme qui introduit un nombre minimal de spill s'il n'y a pas assez de registres dans le processeur.

Supposons qu'il y a R registres disponibles.

L'algorithme de Sethi\_Ullman avec spill commence d'abord par calculer les étiquettes L(u) pour chaque nœud u (voir algo précédent).

*Si*  $L(u1) \geq R$  ET  $L(u2) \geq R$ , alors

- 1/ visiter le fils droit (générer récursivement son code);
- 2/ spiller le fils droit (stocker son résultat en mémoire);
- 3/ visiter le fils gauche (générer récursivement son code);
- 4/ charger le résultat du fils droit;
- 5/ générer le code de l'opération du père;

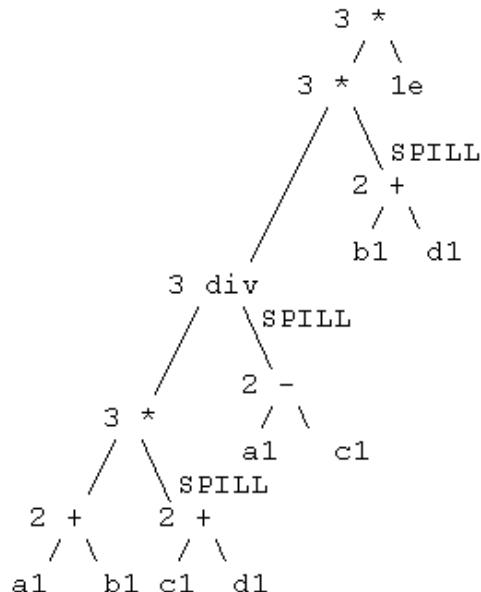
*Si*  $L(u1) < R$  ou  $L(u2) < R$ , continuer comme dans le cas de l'algorithme précédent (cas sans spill), à savoir

```

si  $L(u1) = L(u2)$  alors
{
     $L(u) \leftarrow L(u1) + 1;$  /* un registre supplémentaire est requis
                                pour stocker le résultat intermédiaire
                                d'un des fils */
    Sethi_Ullman(u1); // générer récursivement le code de u1
    Sethi_Ullman(u2); // puis le code de u2, ou l'inverse, c'est pareil */
    générer le code de l'instruction "u1 u.op u2".
}
sinon
{
     $L(u) \leftarrow \text{Max}(L(u1), L(u2));$ 
    // soit u' le fils qui a la plus grande étiquette, u'' celui qui a la plus petite étiquette
    Sethi_Ullman(u'); // générer récursivement le code du noeud qui a la plus grande étiquette L; /* son résultat est dans un registre */
    Sethi_Ullman(u''); // générer ensuite récursivement le code du noeud qui a la plus petite étiquette L;
    /* cet ordre d'évaluation garantit que l'on ne dépasse pas Max(L(u1), L(u2)) registres */
    générer le code de l'instruction "u1 u.op u2";
}
}

```

Voici les sous arbres spilles dans l'arbre précédent en supposant 2 registres disponibles.



Ayant deux registres, on aura besoin de trois temporaires (résultat optimal), comme suit.

```

load  b, R1
load  d, R2
add   R1, R2, R1  /* R1 = b + d */
store temp1, R1  /* temp1 = b + d */

load  a, R1
load  c, R2
sub   R1, R2, R1  /* calcul de a - c */
store temp2, R1  /* temp2 = a - c */

load  c, R1
load  d, R2
add   R1, R2, R1  /* R1 = c + d */
store temp3, R1  /* temp3 = c + d */

load  a, R1
load  b, R2
add   R1, R2, R1  /* R1 = a + b */

load  temp3, R2
mult  R1, R2, R1  /* R1 = (a + b)(c + d) */

load  temp2, R2
div   R1, R2, R1  /* R1 = (a + b)(c + d)/(a - c) */

load  temp1, R2
mult  R1, R2, R1  /* R1 = (a + b)(c + d)/(a - c)(b + d) */

load  e, R2
mult  R1, R2, R1  /* R1 = expression */

```

## gestion de variables temporaires

Au fur et au mesure que l'on génère du code, on aura besoin de variables temporaires. On a le choix entre ajouter ces noms de variables dans la table des symboles (temp1, temp2, ...), ou déclarer un grand tableau temp, que l'on y accédera en l'indexant avec le numéro du temporaire (temp[1], temp[2], etc.).

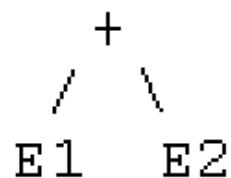
Aussi, on peut utiliser la pile de la machine. En effet, l'algorithme de spill présenté ci-dessus est récursif. Ainsi, au lieu de déclarer un temporaire pour spiller, on peut empiler le résultat intermédiaire dans la pile d'exécution.

### > Solution n°65 (exercice p. 29)

## calcul des constantes

On ajoute un attribut booléen `is_constant` à chaque non-terminal de la grammaire. Ce booléen vaut "vrai" si le non-terminal représente une constante.

Si un non-terminal est une constante, nous n'avons pas besoin de générer son code. Par ex, supposons que nous ayons l'arbre suivant :



supposons que E1 est une constante (vallant 73) et que E2 ne le soit pas. Ainsi, le code généré pour l'arbre ci-dessus est :

<E2.code>

add R <- E2.reg + 73 /\* nous ne générerons pas de code pour E1, il suffit de mettre sa valeur numérique comme argument. Cette valeur est calculée par le compilateur \*/

Notre méthode pour implémenter une telle solution possède en deux étapes.

1. Une première passe avec lex&yacc construit l'arbre abstrait (de bas en haut) et permet de calculer les constantes pendant le parsing comme dans le cas d'une mini-calculatrice (voir les précédents TDs). Cette première passe calcule également les étiquettes L vue dans les questions précédentes.
  2. Une deuxième passe applique l'algo de Sethi Ullman sur l'arbre abstrait de haut en bas.

Passe 1

Les routines sémantiques suivantes ne calculent que les valeurs des constantes et les attributs `is_constant`. Elle n'incluent pas (à tord) les instructions montrant comment construire l'arbre abstrait ni comment calculer les étiquettes L (vues dans les questions précédentes). La construction de l'arbre et le calcul des étiquettes L sont laissés aux étudiants !

Les routines sémantiques suivantes sont exécutées lors du parsing (première passe) de l'expression. Je ne détaille ici que le traitement fait pour évaluer les attributs `is constant` et `val`.

$E \rightarrow E + T$

```
{
  $$.is_constant = $1.is_constant ET $3.is_constant;
  $$.val = $1.val + $3.val;
}
```

$E \rightarrow T$

```
{
  $$.is_constant = $1.is_constant;
  $$.val = $1.val;
}
```

$T \rightarrow T * F$

```
{
  $$.is_constant = $1.is_constant ET $3.is_constant;
  $$.val = $1.val * $3.val;
}
```

$T \rightarrow F$

```
{
  $$.is_constant = $1.is_constant;
  $$.val = $1.val;
}
```

$F \rightarrow CONST$

```
{
  $$.val = $1.val;
  $$.is_constant = true;
}
```

$F \rightarrow ident$

```
{
  $$.is_constant = false;
}
```

$F \rightarrow ( E )$

```
{
  $$.is_constant = $2.is_constant;
  $$.val = $2.val;
}
```

## Passe 2

Cette passe applique l'algorithme récursif de Sethi et Ullman, en parcourant l'arbre de haut en bas (de la racine jusqu'aux feuilles).



```

générer_Code (noeud u)
  Si (u est une feuille et notu.is_constant) alors générer("load R < - u.nom");
  Sinon /* on suppose u1 fils gauche et u2 fils droit */
    générer le code en utilisant l'algorithme de Sethi Ullman uniquement sur les fils qui ne sont pas des constantes.
    Pour les fils constants, prendre uniquement leur valeur comme opérande immédiate lors de la génération de l'opération du père u.
  Fsin

```

## > Solution n°66 (exercice p. 30)

### Actions sémantiques

en pseudo code, pas en syntaxe yacc

On déclare un attribut nom\_res pour chaque non-terminal X. Cet attribut contient le nom de la variable contenant le résultat du sous arbre dont X est la racine.

On suppose aussi un attribut Code pour chaque non terminal.

```

S -> I ';' S { $$ .code = concatener ($1 .code, $3 .code); }
| { $$ .code .init(); //epsilon}

```

```

I -> ident := E
{
  $$ .code .init ($3 .code);
  $$ .code .insert ($2, " = ", $3 .nom_res, ";" \n"); /* on suppose que l'attribut du token
  ident est le nom de l'identifiant */
  $$ .nom_res = strdup ($1 .nom);
}

```

```
I -> E { $$ .code .init ($1 .code); }
```

```

E -> E' +' T
{
  $$ .code .init ($1 .code, $3 .code);
  chaîne_car = new_temp();
  inst_temp = concatener (chaîne_car, " = ", $1 .nom_res, " + ", $3 .nom_res, ";" \n");
  $$ .code .insert (inst_temp);
  $$ .nom_res = strdup (chaîne_car);
}

```

```

E -> T
{
  $$ .code = $1 .code;
  $$ .nom_res = strdup ($1 .nom_res);
}

```

```

 $T \rightarrow T' *' F$ 
{
    $$.code.init($1.code, $3.code);
    chaine_car = new\_temp();
    inst_temp = concatener(chaine_car, " = ", $1.nom_res, "* ", $3.nom_res, "; \n");
    $$.code.insert(inst_temp);
    $$.nom = strdup(chaine_car);
}

 $T \rightarrow F$ 
{
    $$.code = $1.code;
    $$.nom_res = strdup($1.nom_res);
}

 $F \rightarrow ident$ 
{
    ident
    $$.nom_res = strdup($1.nom);
}

 $F \rightarrow CONST$ 
{
    $$.code.init(); // code vide pour CONST
    $$.nom_res = strdup($1.yytext);
}

 $F \rightarrow '(' E ')'$ 
{
    $$.code = $2.code;
    $$.nom_res = strdup($2.nom_res);
}

```

### > Solution n°67 (exercice p. 30)

#### Arbre de dérivation avec code

```

a =3; /* code de I -> ident := CONST */
temp1 = a + 5;
temp2 = temp1 * 2; /* E.nom_res=temp2 */

```

### > Solution n°68 (exercice p. 30)

#### Traduction du for

for ident := E1 to E2 { S }

Je suppose que E a deux attributs:

E.code : contient le code évaluant l'expression E

E.nom\_res : contient le nom de la variable contenant le résultat de l'expression  
I a un attribut I.code.

On suppose que l'expression E2 de fin de boucle n'est évaluée qu'une seule fois à l'entrée de la boucle. En d'autres termes, on interdit que :

- ident le compteur de boucle soit modifié à l'intérieur du corps de boucle
- la valeur de l'expression E2 définissant l'indice maximal d'itérations n'est pas modifiée dans le corps de boucle. Ce n'est pas le cas du langage C par exemple.

Le code à générer dans notre cas est le suivant:

```

< E1.code > /* résultat dans la variable E1.nom_res */
ident.nom = E1.nom_res;
< E2.code > /* résultat dans la variable E2.nom_res */
Etiq_Boucle : if(ident.nom > E2.nom_res) goto FinBoucle;
< S.code >
ident.nom = ident.nom + 1;
goto Etiq_Boucle;
Etiq_FinBoucle :

```

## > Solution n°69 (exercice p. 31)

### Actions sémantiques

```

I -> for ident := E to E '' S ''
{
inst1 = concatener($2.nom, " = ", $4.nom_res, "; \n");

chaine_label_fin_boucle = new_label();
inst2 = concatener("if (", $2.nom, " > ", $6.nom_res, ") goto ", chaine_label_fin_boucle, "; \n");
chaine_label_boucle = new_label();
inst3 = concatener(chaine_label_boucle, " : \n");

inst4 = concatener($2.nom, " = ", $2.nom, " + 1; \n");
inst5 = concatener("goto ", chaine_label_boucle, "; \n");
inst6 = concatener(chaine_label_fin_boucle, " : \n");

$$.code = concatener($4.code, inst1, $6.code, inst3, inst2, $8.code, inst4, inst5, inst6);
}

```

## > Solution n°70 (exercice p. 31)

### code généré

```

a = 1;
temp1 = 5 + 3;
b = temp1;

```

```

L2 : if(b > 10) goto L1;
temp2 = a + 1;
a = temp2;
b = b + 1;
goto L2;

```

```
L1 :
```

## > Solution n°71 (exercice p. 34)

### Exercice 1

Priorité et associativité des opérateurs logiques :

%left or

%left and

%nonassoc not

Et pour if-then-else /\* le premier else rencontré se rattache au dernier if-then \*/

%nonassoc then

%nonassoc else

Il y a deux méthodes d'évaluation d'une expression booléenne/logique :

- ou bien on la calcule comme étant une expression arithmétique;
- ou bien on utilise le flot de contrôle du programme : le résultat d'une expression logique est alors deux étiquettes B.true et B.false.

C'est cette méthode qu'on va utiliser dans ce TD. Cette méthode s'appuie sur les instructions de branchements du processeur.

B.true est un attribut qui contient l'étiquette de la prochaine instruction à exécuter si B est évaluée à vrai;

Réciproquement, B.false est l'attribut qui contient l'étiquette de la prochaine instruction à exécuter si B est évaluée à fausse;

On suppose ce qui suit :

- le terminal op a un attribut "comp" qui sert à déterminer le type d'opérateur de comparaison utilisé parmi {<, >, ==, !=}.
- le non-terminal B possède les attributs suivants:
- Deux étiquettes B.true et B.false.
- Un code généré B.code. Ce code génère un branchement vers B.true si l'expression est vraie, et vers B.false dans le cas contraire.
- le non-terminal E a un attribut "code" qui contient le code évaluant l'expression, et un attribut "nom\_res" qui contient le nom de la variable contenant le résultat de \$E\$.
- On dispose d'une fonction nouvelle\_etiquette() qui fournit un nouveau nom d'étiquette non utilisé à chaque appel.
- On suppose que la fonction "concat" permet de concaténer des chaînes de caractères : elle est utilisée pour la génération de code (instructions).

Les routines sémantiques ci-dessous ne font pas partie d'une traduction dirigée par la syntaxe S-attribuée. En effet, l'attribut B.code est synthétisé, par contre les attributs B.true et B.false sont hérités.

<i>Traduction de if – then</i>		
	< B.code >	- > vers B.vrai - > vers B.faux
B.vrai :	< S.code >	
B.faux :	...	

Dans le cas d'un if-then-else, I.suiv est l'étiquette de l'instruction juste après le if-then-else.



*Traduction de if – then – else*

	$< B.code >$	$\rightarrow$ vers $B.vrai$
		$\rightarrow$ vers $B.faux$
$B.vrai :$	$< S1.code >$	
	<i>br I.suiv</i>	
$B.faux :$	$< S2.code >$	
$I.suiv :$	...	

Les règles sémantiques suivantes ne sont guère destinées à une traduction dirigées par la syntaxe S-attribuée (yacc). Ceci, car elles calculent des attributs hérités, exigeant un parcours de l'arbre syntaxique de haut en bas, et non pas de bas en haut comme le fait yacc.

```
/* ici les attributs B.true et B.false sont hérités, alors que l'attribut I.code est synthétisé */
I -> if (B) then {S}
{
    B.true = nouvelle_etiquette();
    B.false = nouvelle_etiquette();
    etiquette1 = concat(B.true, ':');
    etiquette2 = concat(B.false, ':');
    I.code = concat(B.code, etiquette1, S.code, etiquette2);
}

I -> if (B) then {S1} else {S2}
{
    B.true = nouvelle_etiquette();
    B.false = nouvelle_etiquette();
    I.suiv = nouvelle_etiquette();
    etiquette1 = concat(B.true, ':');
    etiquette2 = concat(B.false, ':');
    etiquette3 = concat(I.suiv, ':');
    branch = concat("br", I.suiv);

    I.code = concat(B.code, etiquette1, S1.code, branch, etiquette2, S2.code, etiquette3); /* générer le code */
}
```

## > Solution n°72 (exercice p. 34)

### génération de B.code

$B \rightarrow B1 \text{ or } B2$  se traduit en

$\quad < B1.code >$   
 $B1.faux : < B2.code >$

$B \rightarrow B1 \text{ or } B2$

```
{  
    B1.true = B.true; /* si B1 est vrai, alors B = true */  
    B1.false = nouvelle_etiquette(); /* si B1 est faux, alors B = B2 */  
    B2.true = B.true;  
    B2.false = B.false;  
    B.code = concat(B1.code, generer(B1.false, ':'), B2.code);  
}
```

$B \rightarrow B1 \text{ and } B2$  se traduit en

$\quad < B1.code >$   
 $B1.vrai : < B2.code >$

$B \rightarrow B1 \text{ and } B2$

```
{  
    B1.false = B.false; /* si B1 est faux, alors B = false */  
    B1.true = nouvelle_etiquette(); /* si B1 est vrai, alors B = B2 */  
    B2.true = B.true;  
    B2.false = B.false;  
    etiq_temp = concat(B1.true, ' :')  
    B.code = concat(B1.code, etiq_temp, B2.code);  
}
```

```
B -> not B1
{
  B1.false = B.true;
  B1.true = B.false;
  B.code = B1.code;
}
```

```
B -> (B1)
{
  B1.false = B.false;
  B1.true = B.true;
  B.code = B1.code;
}
```

*B -> E1 op E2 :*  
*exemple : a < b se traduit en*  
*si a < b alors goto B.true*  
*goto B.false*

```
B -> E1 op E2
{
  inst1 = concat("if", E1.nom_res, op.comp, E2.nom_res, "goto", B.true);
  inst2 = concat("goto", B.false);
  B.code = concat(E1.code, E2.code, inst1, inst2);
}
```

```
B -> true
{
  B.code = concat("goto", B.true);
}
```

```
B -> false
{
  B.code = concat("goto", B.false);
}
```

### > Solution n°73 (exercice p. 34)

Avec les actions sémantiques précédentes, l'expression a<b ou c<d et e<f se traduit en

```

    si a < b goto Etiquette_true
    goto L1
L1 :  si c < d goto L2
      goto Etiquette_false
L2 :  si e < f goto Etiquette_true
      goto Etiquette_false
...
Etiquette_true :
...
Etiquette_false :

```

On pourrait faire mieux : en effet, si on laisse l'exécution du code généré suivre son cours, alors l'expression précédente se réduit en :

exemple :  $a < b$  se traduit en

*si a >= b alors goto B.false*

```

B -> E1 op E2
{
  B.code = concatr("if", E1.nom_res, inverser(op.com), E2.nom_res, "goto", B.false);
}

B -> true
{
  /* laisser l'execution continuer */
}
B -> false
{
  B.code = concat("goto", B.false);
}

```

Cette méthode de génération de code pour une expression logique B nous oblige dorénavant à placer le code de la branche vrai juste après le code qui évalue B. Avec la méthode des branchements standards (voir question précédente), on avait deux étiquettes (vrai et faux), on pouvait placer les deux branches d'un if dans n'importe quel ordre. Or avec la méthode actuelle, la branche vraie doit apparaître juste après B.code.

Les autres règles sémantiques peuvent rester inchangées. Néanmoins, elles génèrent des étiquettes caduques (étiquettes des branches vrai).

## > Solution n°74 (exercice p. 34)

### code trois adresses

```

i = 0;
    if (i > 10) goto fin;
boucle :
    a = i + 1;
    b = i + 1;
    c = i + 2;
    tmp = a + b;
    i = i + 1;
    if (i <= 10) goto boucle;
fin :

```

Bloc de base et CFG : voir réponse de la question suivante

### > Solution n°75 (exercice p. 34)

#### Bloc de base et CFG

*B1* : *i = 0;*  
*if (i > 10) goto B3;*

*B2* : *a = i + 1;*  
*b = i + 1;*  
*c = i + 2;*  
*tmp = a + b;*  
*i = i + 1;*  
*if(i <= 10)gotoB2;*

*B3* : *exit*

### > Solution n°76 (exercice p. 35)

#### élimination des sous-expressions communes (common sub-expression elimination)

*B1* : *rien à faire*

*B2*  
*a = i + 1;*  
*b = a;*  
*c = i + 2;*  
*tmp = a + b;*  
*i = a;*

### > Solution n°77 (exercice p. 35)

#### élimination de code mort (dead code elimination)

Le code mort est tout bout de programme qui ne sert à rien! En d'autres termes, il ne contribue pas au résultat final calculé par le programme. Deux cas de code mort:

1. les blocs de base inaccessibles dans le CFG : ce sont les blocs de base vers lesquels on ne se branche jamais. Il est clair qu'on peut les enlever non pas pour accélérer le programme (puisque de toute façon, ils ne sont pas

exécutés), mais pour réduire la taille de code généré.

2. les instructions produisant des résultats intermédiaires qui ne contribuent pas aux résultats finaux du programme peuvent être éliminées. Les informations sur les flots de données nous renseignent à ce sujet.

Dans notre cas, si aucune des variables calculées par le programme n'est ni lue ni affichée ni produite en sortie, on peut éliminer tout ce code ! Croyez moi, les compilateurs ne se gênent pas pour le faire.

Supposons que a et i sont des variables lues après la boucle, mais pas les variables c et tmp.

Donc, en faisant une première étape d'élimination de code mort, on obtient :

B2 :

```
a=i+1;  
b=a;  
i=a
```

Lors d'une deuxième étape, on remarque que l'instruction "b=a" peut aussi être éliminée.