Université Laval
Faculté des sciences et de génie
Département d'informatique et de génie logiciel
IFT-3101

Danny Dubé
Version : Hiver 2014

Exercices reliés au chapitre 6

Exercices

Voici les exercices que je recommande de faire :

— **Exercice 6.2.1**. (Exercice 8.1 dans la 1ère édition.)

— **Exercice 6.2.2** *i* et *ii*. (Dans la 1ère édition, l'exercice 8.2 est similaire.)
Note : au numéro i, assumez que le premier élément d'un tableau est à l'indice zéro.

— **Exercice 6.3.1**.
Note : assumez que les variables de type int *et* float *ont une taille de 4 octets.*

— **Exercices 6.4.1** et **6.4.2**.

— **Exercice 6.4.3**. (Dans la 1ère édition, l'exercice 8.3 est similaire.)
Note : portez bien attention à la différence entre `t` *et* `t.elem` *lorsque* `t` *est un type tableau (array).*
Par exemple, si `t` *est le type "tableau de int" (avec* `sizeof(int) == 4` *octets), alors* `t.elem` *vaut "int" ;*
De plus, notez que si `t` *est le type "tableau de (tableau de int)",* `t.elem` *vaut "tableau de int" et* `t.elem.elem` *vaut "int".*

— **Exercice 6.4.6**.

— **Exercice 6.4.8**.

— **Exercice 6.5.1**.
Note : utilisez la grammaire de la figure 6.20, en vous inspirant du morceau de système de traduction présenté à la figure 6.27 pour traiter le typage.

— **Exercice 6.6.1**. (Dans la 1ère édition, l'exercice 8.14 est similaire.)
Note : donnez le code "régulier" ***et*** *le code à court-circuit (en vous inspirant de la figure 6.36)*
Note : pour la construction ***repeat ... while ...****, assumez une sémantique similair à celle de* ***do ... while*** *en C, i.e. le corps de la boucle est toujours exécuté au moins une fois.*

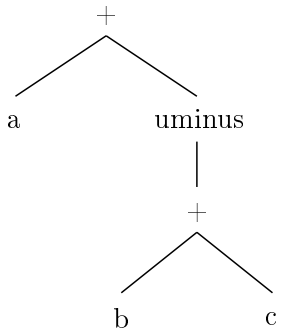
— **Exercice 6.6.3**.

— (Dans la 1ère édition, les exercices sur le typage sont 6.1, 6.2, 6.3, 6.5 et 6.8. Ils n'ont pas vraiment de correspondance avec ceux de la nouvelle édition.)

Réponses

6.2.1

— (a)



L'opérateur *uminus* correspond au moins unaire (négation d'un nombre).

— (b)

	Op	**Arg 1**	**Arg 2**	**Résultat**
0	+	b	c	t_1
1	uminus	t_1		t_2
2	+	a	t_2	t_3

— (c)

	Op	**Arg 1**	**Arg 2**
0	+	b	c
1	uminus	(0)	
2	+	a	(1)

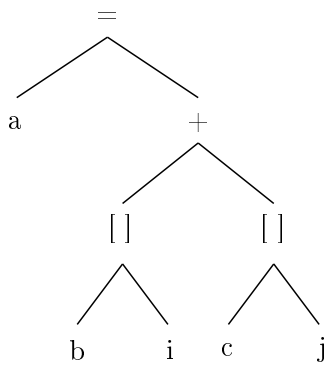
— (d)

instruction	
n	(0)
n + 1	(1)
n + 2	(2)

	Op	**Arg 1**	**Arg 2**
0	+	b	c
1	uminus	(0)	
2	+	a	(1)

6.2.2 (i)

— (a)



— (b) Nous assumons ici que la table des symboles contient un attribut *len* qui indique la taille en octets de chaque élément du tableau (si le type du symbole est un type tableau). Nous utilisons de plus l'opérateur *deref*, qui correspond à déréférencer un pointeur ("aller à l'adresse de", correspond à l'opérateur préfixe unaire $*$ en C).

	Op	**Arg 1**	**Arg 2**	**Résultat**
0	*	b.type.elem.width	i	t_1
1	+	b	t_1	t_2
2	*	c.type.elem.width	j	t_3
3	+	c	t_3	t_4
4	deref	t_2		t_5
5	deref	t_4		t_6
6	+	t_5	t_6	a

— (c)

	Op	**Arg 1**	**Arg 2**
0	*	b.type.elem.width	i
1	+	b	(0)
2	*	c.type.elem.width	j
3	+	c	(2)
4	deref	(1)	
5	deref	(3)	
6	+	(4)	(5)
7	:=	a	(6)

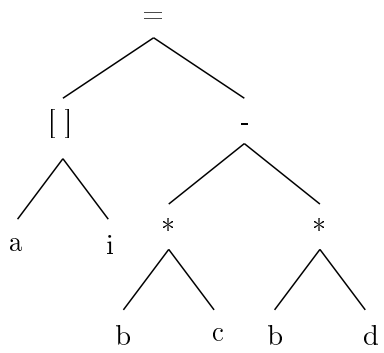
— (d)

instruction	
n	(0)
n + 1	(1)
n + 2	(2)
n + 3	(3)
n + 4	(4)
n + 5	(5)
n + 6	(6)
n + 7	(7)

	Op	Arg 1	Arg 2
0	*	b.type.elem.width	i
1	+	b	(0)
2	*	c.type.elem.width	j
3	+	c	(2)
4	deref	(1)	
5	deref	(3)	
6	+	(4)	(5)
7	:=	a	(6)

6.2.2 (ii)

— (a)



— (b) *Note : nous utilisons l'instruction* `store x y` *pour représenter l'assignation à travers un pointeur, où x est le pointeur et y la valeur à assigner (correspond à* `*x = y` *en C)*

	Op	Arg 1	Arg 2	Résultat
0	*	b	c	t_1
1	*	b	d	t_2
2	-	t_1	t_2	t_3
3	*	a.type.elem.width	i	t_4
4	+	a	t_4	t_5
5	Store	t_5	t_3	

— (c)

	Op	Arg 1	Arg 2
0	*	b	c
1	*	b	d
2	-	(0)	(1)
3	*	a.type.elem.width	i
4	+	a	(3)
5	Store	(4)	(2)

— (d)

instruction	
n	(0)
n + 1	(1)
n + 2	(2)
n + 3	(3)
n + 4	(4)
n + 5	(5)

	Op	Arg 1	Arg 2
0	*	b	c
1	*	b	d
2	-	(0)	(1)
3	*	a.type.elem.width	i
4	+	a	(3)
5	Store	(4)	(2)

6.3.1

On note que le `record` p a une taille de 8 octets, et le `record` q, une taille de 12 octets. On trouve donc :

- x : adresse 0, taille 4, type **float**
- p : adresse 4, taille 8, type **record** { x : **float** ; y : **float** }
  - ◇ x : adresse 0, taille 4, type **float**
  - ◇ y : adresse 4, taille 4, type **float**
- q : adresse 12, taille 12, type **record** { tag : **int** ; x : **float** ; y : **float** }
  - ◇ *tag* : adresse 0, taille 4, type **int**
  - ◇ x : adresse 4, taille 4, type **float**
  - ◇ y : adresse 8, taille 4, type **float**

Notez que les adresses dans un *record* sont relatives au début de leur *record*.

6.4.1

— (a)

Production	Règles sémantiques
$E \rightarrow E_1 * E_2$	$E.addr := \textbf{new } Temp()$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.addr \text{ ‘=’ } E_1.addr \text{ ‘*’ } E_2.addr)$

— (b)

Production	**Règles sémantiques**
$E \rightarrow +E_1$	$E.addr := \textbf{new } Temp()$ $E.code := E_1.code \parallel gen(E.addr \text{ ‘=’ } E_1.addr)$

6.4.2

— (a)

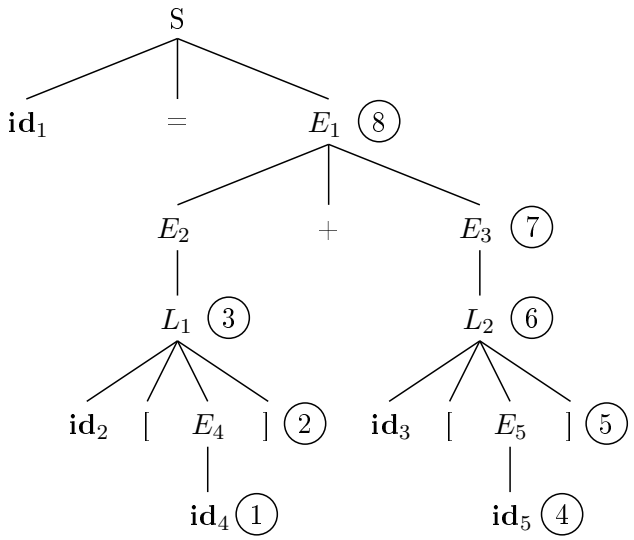
$$E \rightarrow E_1 * E_2 \quad \{ \quad E.addr := \textbf{new } Temp() \quad gen(E.addr \text{ ‘=’ } E_1.addr \text{ ‘*’ } E_2.addr) \quad \}$$

— (b)

$$E \rightarrow +E_1 \quad \{ \quad E.addr := \textbf{new } Temp() \quad gen(E.addr \text{ ‘=’ } E_1.addr) \quad \}$$

6.4.3 (a)

Cette expression correspond à l'arbre syntaxique suivant :



avec la table des symboles suivante :

Symbole	**Nom**
$\textbf{id}_1$	x
$\textbf{id}_2$	a
$\textbf{id}_3$	b
$\textbf{id}_4$	i
$\textbf{id}_5$	j

Les règles sémantiques sont toutes exécutées à la fin d'une production ; nous effectuons donc un parcours en profondeur d'abord (les annotations numériques indiquent dans quel ordre les règles sémantiques seront exécutées).

Notez que le manuel utilise `top.get(id.lexeme)` pour retrouver un symbole dans la table des symboles ; pour des raisons de brièveté, nous nous contenterons ici d'écrire `id` et supposerons implicitement qu'une implémentation concrète effectuerait les étapes nécessaires pour obtenir l'information dans la table des symboles.

Voici une trace du parcours :

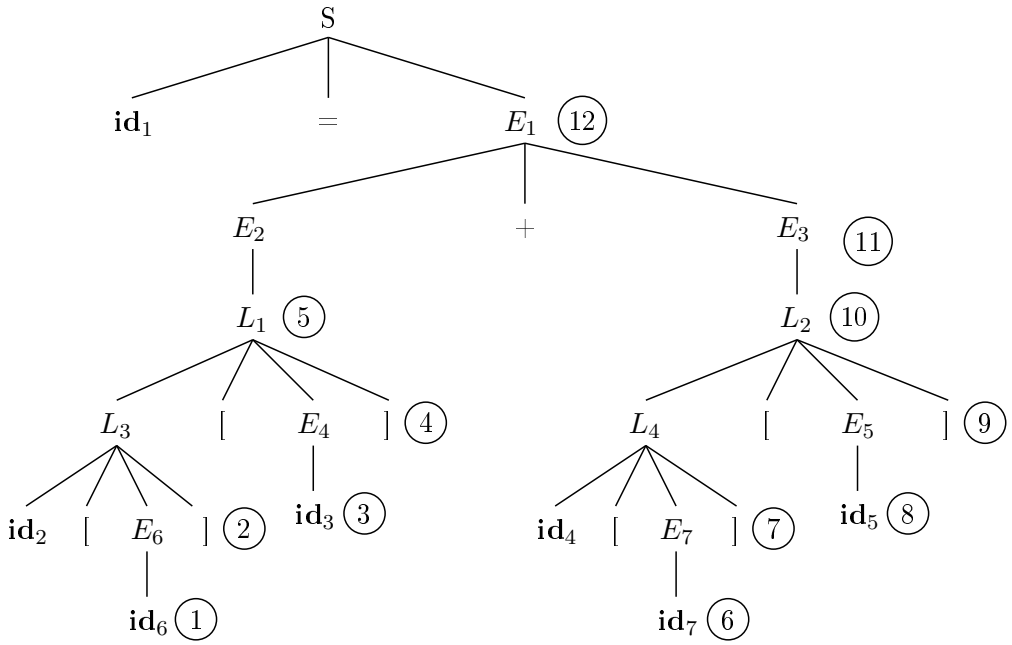
Étape	**Attributs et temporaires**	**Code généré**
1	$E_4.addr = \mathbf{id}_4.place$	
2	$L_1.array = \mathbf{id}_2$ $L_1.type = L_1.array.type.elem$ $= \mathbf{id}_2.type.elem$ $L_1.addr = \mathbf{new}\ Temp() = t_1$	$t_1 = E_4.addr * L_1.type.width$ **c-à-d** $t_1 = \mathbf{id}_4.place * \mathbf{id}_2.type.elem.width$
3	$E_2.addr = \mathbf{new}\ Temp() = t_2$	$t_2 = L_1.array.base[L_1.addr]$ **c-à-d** $t_2 = \mathbf{id}_2.base[\ t_1\ ]$
4	$E_5.addr = \mathbf{id}_5.place$	
5	$L_2.array = \mathbf{id}_3$ $L_2.type = L_2.array.type.elem$ $= \mathbf{id}_3.type.elem$ $L_2.addr = \mathbf{new}\ Temp() = t_3$	$t_3 = E_5.addr * L_2.type.width$ **c-à-d** $t_3 = \mathbf{id}_5.place * \mathbf{id}_3.type.elem.width$
6	$E_3.addr = \mathbf{new}\ Temp() = t_4$	$t_4 = L_2.array.base[L_2.addr]$ **c-à-d** $t_4 = \mathbf{id}_3.base[\ t_3\ ]$
7	$E_1.addr = \mathbf{new}\ Temp() = t_5$	$t_5 = E_2.addr\ +\ E_3.addr$ **c-à-d** $t_5 = t_2 + t_4$
8		$\mathbf{id}_1.place = t_5$

Au final, nous avons généré le code suivant :

$$
\begin{aligned}
t_1 &= \mathbf{id}_4.place * \mathbf{id}_2.type.elem.width \\
t_2 &= \mathbf{id}_2.base[t_1] \\
t_3 &= \mathbf{id}_5.place * \mathbf{id}_3.type.elem.width \\
t_4 &= \mathbf{id}_3.base[t_3] \\
t_5 &= t_2 + t_4 \\
\mathbf{id}_1.place &= t_5
\end{aligned}
$$

6.4.3 (b)

Cette expression correspond à l'arbre syntaxique suivant :



avec la table des symboles suivante :

Symbole	**Nom**
$\mathbf{id}_1$	x
$\mathbf{id}_2$	a
$\mathbf{id}_3$	j
$\mathbf{id}_4$	b
$\mathbf{id}_5$	j
$\mathbf{id}_6$	i
$\mathbf{id}_7$	i

Voici une trace du parcours :

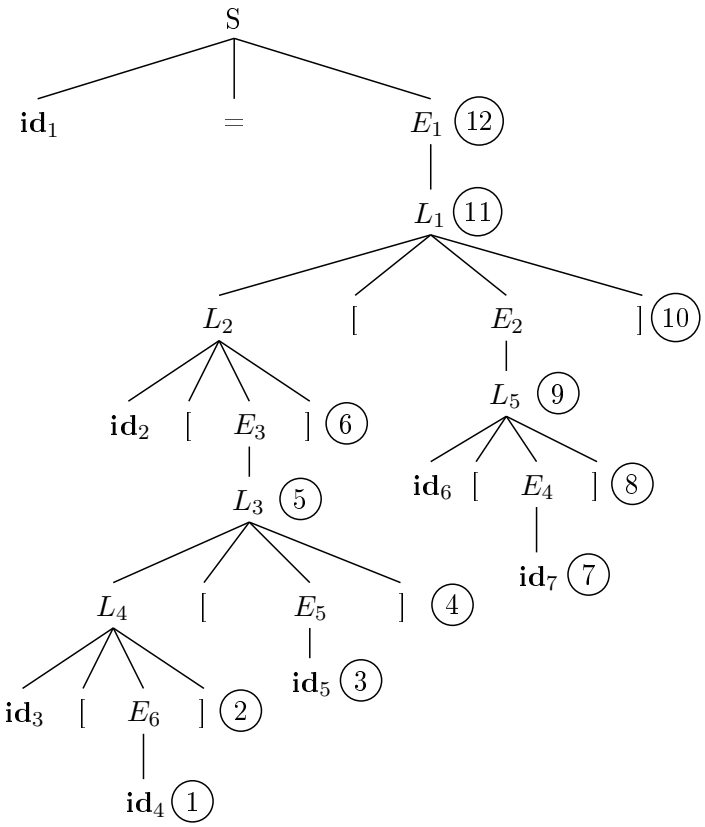
Étape	**Attributs et temporaires**	**Code généré**
1	$E_6.addr = \mathbf{id}_6.place$	
2	$L_3.array = \mathbf{id}_2$ $L_3.type = L_3.array.type.elem$ $= \mathbf{id}_2.type.elem$ $L_3.addr = \mathbf{new}\ Temp() = t_1$	$t_1 = \mathbf{id}_6.place * \mathbf{id}_2.type.elem.width$
3	$E_4.addr = \mathbf{id}_3.place$	
4	$L_1.array = \mathbf{id}_2$ $L_1.type = L_3.type.elem$ $= \mathbf{id}_2.type.elem.elem$ $t = \mathbf{new}\ Temp() = t_2$ $L_1.addr = \mathbf{new}\ Temp() = t_3$	$t_2 = \mathbf{id}_3.place * \mathbf{id}_2.type.elem.elem.width$ $t_3 = t_1 + t_2$
5	$E_2.addr = \mathbf{new}\ Temp() = t_4$	$t_4 = \mathbf{id}_2.base[t_3]$
6	$E_7.addr = \mathbf{id}_7.place$	
7	$L_4.array = \mathbf{id}_4$ $L_4.type = L_4.array.type.elem$ $= \mathbf{id}_4.type.elem$ $L_4.addr = \mathbf{new}\ Temp() = t_5$	$t_5 = \mathbf{id}_7.place * \mathbf{id}_4.type.elem.width$
8	$E_5.addr = \mathbf{id}_5.place$	
9	$L_2.array = \mathbf{id}_4$ $L_2.type = L_4.type.elem$ $= \mathbf{id}_4.type.elem.elem$ $t = \mathbf{new}\ Temp() = t_6$ $L_2.addr = \mathbf{new}\ Temp() = t_7$	$t_6 = \mathbf{id}_5.place * \mathbf{id}_4.type.elem.elem.width$ $t_7 = t_5 + t_6$
10	$E_3.addr = \mathbf{new}\ Temp() = t_8$	$t_8 = \mathbf{id}_4.base[t_7]$
11	$E_1.addr = \mathbf{new}\ Temp() = t_9$	$t_9 = t_4 + t_8$
12		$\mathbf{id}_1.place = t_9$

Au final, nous avons généré le code suivant :

$$
\begin{aligned}
t_1 &= \mathbf{id}_6.place * \mathbf{id}_2.type.elem.width \\
t_2 &= \mathbf{id}_3.place * \mathbf{id}_2.type.elem.elem.width \\
t_3 &= t_1 + t_2 \\
t_4 &= \mathbf{id}_2.base[t_3] \\
t_5 &= \mathbf{id}_7.place * \mathbf{id}_4.type.elem.width \\
t_6 &= \mathbf{id}_5.place * \mathbf{id}_4.type.elem.elem.width \\
t_7 &= t_5 + t_6 \\
t_8 &= \mathbf{id}_4.base[t_7] \\
t_9 &= t_4 + t_8 \\
\mathbf{id}_1.place &= t_9
\end{aligned}
$$

6.4.3 (c)

Cette expression correspond à l'arbre syntaxique suivant :



avec la table des symboles suivante :

Symbole	**Nom**
$\mathbf{id}_1$	x
$\mathbf{id}_2$	a
$\mathbf{id}_3$	b
$\mathbf{id}_4$	i
$\mathbf{id}_5$	j
$\mathbf{id}_6$	c
$\mathbf{id}_7$	k

Voici une trace du parcours :

Étape	**Attributs et temporaires**	**Code généré**
1	$E_6.addr = \mathbf{id}_4.place$	
2	$L_4.array = \mathbf{id}_3$ $L_4.type = L_4.array.type.elem$ $= \mathbf{id}_3.type.elem$ $L_4.addr = \mathbf{new}\ Temp() = t_1$	$t_1 = \mathbf{id}_4.place * \mathbf{id}_3.type.elem.width$
3	$E_5.addr = \mathbf{id}_5.place$	
4	$L_3.array = L_4.array$ $= \mathbf{id}_3$ $L_3.type = L_4.type.elem$ $= \mathbf{id}_3.type.elem.elem$ $t = \mathbf{new}\ Temp() = t_2$ $L_3.addr = \mathbf{new}\ Temp() = t_3$	$t_2 = \mathbf{id}_5.place * \mathbf{id}_3.type.elem.elem.width$ $t_3 = t_1 + t_2$
5	$E_3.addr = \mathbf{new}\ Temp() = t_4$	$t_4 = \mathbf{id}_3.base[t_3]$
6	$L_2.array = \mathbf{id}_2$ $L_2.type = \mathbf{id}_2.type.elem$ $L_2.addr = \mathbf{new}\ Temp() = t_5$	$t_5 = t_4 * \mathbf{id}_2.type.elem.width$
7	$E_4.addr = \mathbf{id}_7.place$	
8	$L_5.array = \mathbf{id}_6$ $L_5.type = L_5.array.type.elem$ $= \mathbf{id}_6.type.elem$ $L_5.addr = \mathbf{new}\ Temp() = t_6$	$t_6 = \mathbf{id}_7.place * \mathbf{id}_6.type.elem.width$
9	$E_2.addr = \mathbf{new}\ Temp() = t_7$	$t_7 = \mathbf{id}_6.base[t_6]$
10	$L_1.array = L_2.array$ $= \mathbf{id}_2$ $L_1.type = L_2.type.elem$ $= \mathbf{id}_2.type.elem.elem$ $t = \mathbf{new}\ Temp() = t_8$ $L_1.addr = \mathbf{new}\ Temp() = t_9$	$t_8 = t_7 * \mathbf{id}_2.type.elem.elem.width$ $t_9 = t_5 + t_8$
11	$E_1.addr = \mathbf{new}\ Temp() = t_{10}$	$t_{10} = \mathbf{id}_2.base[t_9]$
12		$\mathbf{id}_1.place = t_{10}$

Au final, nous avons généré le code suivant :

$$
\begin{aligned}
t_1 &= \mathbf{id}_4.place * \mathbf{id}_3.type.elem.width \\
t_2 &= \mathbf{id}_5.place * \mathbf{id}_3.type.elem.elem.width \\
t_3 &= t_1 + t_2 \\
t_4 &= \mathbf{id}_3.base[t_3] \\
t_5 &= t_4 * \mathbf{id}_2.type.elem.width \\
t_6 &= \mathbf{id}_7.place * \mathbf{id}_6.type.elem.width \\
t_7 &= \mathbf{id}_6.base[t_6] \\
t_8 &= t_7 * \mathbf{id}_2.type.elem.elem.width \\
t_9 &= t_5 + t_8 \\
t_{10} &= \mathbf{id}_2.base[t_9] \\
\mathbf{id}_1.place &= t_{10}
\end{aligned}
$$

6.4.6

La formule pour déterminer la position de l'élément $[i, j]$ est $(i-1) * 20 + j - 1$ (la soustraction par 1 est nécessaire car les indices commencent à 1) ; il reste alors à multiplier par 4 puisque chaque item du tableau a une taille de 4 octets.

— (a) $4 * (3 * 20 + 4) = 256$
— (b) $4 * (9 * 20 + 7) = 748$
— (c) $4 * (2 * 20 + 16) = 224$

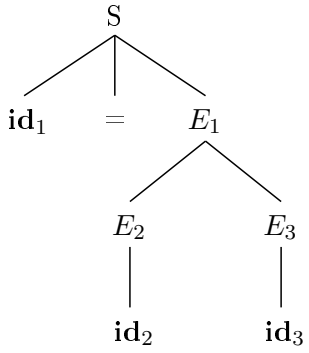
6.4.8

Il y a 5 éléments dans la deuxième dimension (j va de 0 à 4). Il y en a 6 dans la troisième (k va de 5 à 10). Chaque élément de base (chaque réel) occupe 8 octets. La formule pour déterminer la position de l'élément $A[i, j, k]$ est donc $(((i-1) \times 5 \ + \ j) \times 6 \ + \ k - 5) \times 8$.

— (a) $(((3-1) \times 5 \ + \ 4) \times 6 \ + \ 5 - 5) \times 8 = 672$
— (b) $(((1-1) \times 5 \ + \ 2) \times 6 \ + \ 7 - 5) \times 8 = 112$
— (c) $(((4-1) \times 5 \ + \ 3) \times 6 \ + \ 9 - 5) \times 8 = 896$

6.5.1

— (a) Voici l'arbre de dérivation de la chaine :



On applique les règles :

Production	**Application des règles sémantiques**
$E_2 \rightarrow \mathbf{id}_2$	$E_2.addr = \mathbf{id}_2.place$
	$E_2.type = \mathbf{short}$
$E_3 \rightarrow \mathbf{id}_3$	$E_3.addr = \mathbf{id}_3.place$
	$E_3.type = \mathbf{char}$
$E_1 \rightarrow E_2 + E_3$	$E_1.type = max(\mathbf{short}, \mathbf{char}) = \mathbf{int}$
	$a_1 = widen(\mathbf{id}_2.place, \mathbf{short}, \mathbf{int})$
	$= \begin{cases} t_1 = \mathbf{new}\, Temp(); \\ gen(t_1\ `=\text{'}\ `(\mathbf{int})\text{'}\ \mathbf{id}_2); \\ \mathbf{return}\ t_1 \end{cases}$
	$a_2 = widen(\mathbf{id}_3.place, \mathbf{char}, \mathbf{int})$
	$= \begin{cases} t_2 = \mathbf{new}\ Temp(); \\ gen(t_2\ `=\text{'}\ `(\mathbf{int})\text{'}\ \mathbf{id}_3); \\ \mathbf{return}\ t_2 \end{cases}$
	$E_1.addr = \mathbf{new}\, Temp() = t_3$
	$gen(t_3\text{`='}a_1\ `+\text{'}\ a_2)$
$S \rightarrow \mathbf{id}_1 = E_1$	$S.addr = \mathbf{new}\, Temp() = t_4$
	$gen(t_4\text{`='}\text{`}(float)\text{'}t_3)$
	$gen(\mathbf{id}_1.place\text{`='}t_4)$

Le code résultant est :

$$
\begin{aligned}
t_1 &= (\mathbf{int})\ \mathbf{id}_2 \\
t_2 &= (\mathbf{int})\ \mathbf{id}_3 \\
t_3 &= t_1 + t_2 \\
t_4 &= (\mathbf{float})\ t_3 \\
\mathbf{id}_1.place &= t_4
\end{aligned}
$$

— (b) Ce numéro est identique un numéro (a) à l'exception de la conversion finale de **int** vers **float**, qui n'est pas nécessaire ici. Le code généré est :

$$\begin{aligned} t_1 &= (\textbf{int})\ \textbf{id}_2 \\ t_2 &= (\textbf{int})\ \textbf{id}_3 \\ t_3 &= t_1 + t_2 \\ \textbf{id}_1.\mathit{place} &= t_3 \end{aligned}$$

— (c) Le principe est le même qu'en (a), à l'exception que cette fois il y a deux fois l'addition **char** + **short**. Le code généré est :

$$\begin{aligned} t_1 &= (\textbf{int})\ \textbf{id}_2 \\ t_2 &= (\textbf{int})\ \textbf{id}_3 \\ t_3 &= t_1 + t_2 \\ t_4 &= (\textbf{int})\ \textbf{id}_4 \\ t_5 &= (\textbf{int})\ \textbf{id}_5 \\ t_6 &= t_4 + t_5 \\ t_7 &= t_3 * t_6 \\ t_8 &= (\textbf{float})\ t_7 \\ \textbf{id}_1.\mathit{place} &= t_8 \end{aligned}$$

6.6.1 (code régulier)

— (a) Le code à générer a cette allure :

```
debut_boucle :
    S1.code
    B.code
    if (B.place != 0) goto debut_boucle
```

La définition orientée-syntaxe correspondante est :

Productions	**Règles sémantiques**
$S \rightarrow \textbf{repeat}\ S_1\ \textbf{while}\ B$	$S.lbl := newlabel()$ $S.code := label(S.lbl) \parallel S_1.code \parallel B.code \parallel$ $gen(\text{`if ('}B.place\text{`}\neq 0\text{)''goto'}S.lbl)$

— (b) Le code à générer a cette allure (il existe d'autres possibilités) :

```
S1.code
goto l_verification
l_debut:
    S3.code
    S2.code
l_verification:
    B.code
    if (B.place != 0) goto l_debut
```

La définition orientée-syntaxe correspondante est :

Productions	**Règles sémantiques**
$S \rightarrow$ **for** $(S_1\ ;\ B\ ;\ S_2)\ S_3$	$S.ld := newlabel()$ $S.lv := newlabel()$ $S.code := S_1.code \parallel gen(\text{`goto'}\ S.lv) \parallel$ $label(S.ld) \parallel S_3.code \parallel S_2.code \parallel$ $label(S.lv) \parallel B.code \parallel$ $gen(\text{`if (`}B.place\text{`}\neq 0\text{)'`goto'}\ S.ld)$

6.6.1 (code à court-circuit)

— (a)

Productions	**Règles sémantiques**
$S \rightarrow$ **repeat** S_1 **while** B	$S_1.next := newlabel()$ $B.true := newlabel()$ $B.false := S.next$ $S.code := label(B.true) \parallel S_1.code \parallel$ $label(S_1.next) \parallel B.code \parallel$ $label(B.false)$

— (b)

Productions	**Règles sémantiques**
$S \rightarrow$ **for** $(S_1; B\ ; S_2)\ S_3$	$begin := newlabel()$ $B.true := newlabel()$ $B.false := S.next$ $S_2.next := begin$ $S_1.next := begin$ $S_3.next := newlabel()$ $S.code := S_1.code \parallel label(begin) \parallel B.code \parallel$ $label(B.true) \parallel S_3.code \parallel$ $label(S_3.next) \parallel S_2.code \parallel$ $gen(\text{`}goto\text{'}\ begin)$

6.6.3

Disons que notre opérateur *ou exclusif* s'écrit avec `^`. Malheureusement, afin d'avoir un code aussi rapide que celui généré par les règles de la figure 6.37, il faudrait qu'une des deux sous-expressions soit dupliquée. Or, le reste des règles ne produit qu'une seule version du code de chaque sous-expression. Nous devons donc nous contenter d'un code moins performant. De plus, pour des programmes particuliers, la duplication pourrait produire un code compilé exponentiellement plus volumineux que le code source.

PRODUCTION	RÈGLES SÉMANTIQUES
$B \rightarrow B_1$ `^` B_2	$B.tmp =$ **new** $Temp()$ $B_1.true = newlabel()$ $B_1.false = newlabel()$ $B_2.true = newlabel()$ $B_2.false = newlabel()$ $B.ccode = gen(B.tmp$ '=' '0') || $B_1.ccode$ || $label(B_1.true)$ || $gen(B.tmp$ '=' '1' '-' $B.tmp)$ || $label(B_1.false)$ || $B_2.ccode$ || $label(B_2.true)$ || $gen(B.tmp$ '=' '1' '-' $B.tmp)$ || $label(B_2.false)$ || $gen($'`if`' $B.tmp$ '=' '1' '`goto`' $B.true)$ || $gen($'`goto`' $B.false)$