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

Danny Dubé
Version : Automne 2020

Exercices reliés au chapitre 5

Exercices

Voici les exercices que je recommande de faire :

— **Exercice 5.1.1**. (Exercice 5.1 dans la 1ère édition.)

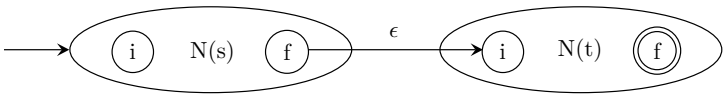
— **Exercice 5.2.1**
Note : la question demande de lister toutes les possibilités ; contentez-vous d'en donner 3.

— **Exercice 5.2.2**. (Similaire à l'exercice 5.9 dans la 1ère édition.)

— **Exercice 5.2.3**.

— **Exercices 5.2.4** et **5.2.5**. (Exercice 5.8 dans la 1ère édition.)

— **Exercice 5.2.6**.
Note : pour la concaténation de deux expressions régulières, il n'est pas nécessaire de procéder à la fusion d'états comme suggéré dans le manuel ; il suffit d'ajouter une epsilon-transition comme ceci :



— **Exercice 5.3.1**. (Exercice 5.6 dans la 1ère édition.)
***Note** : assumez qu'additionner un int avec un float donne un float*
***Note** : vous n'avez pas besoin d'utiliser l'opérateur **intToFloat** mentionné dans l'énoncé.*

— (Optionnel) **Exercice 5.3.2**. (Exercice 5.4 dans la 1ère édition.)
***Note** : Vous ne pouvez pas assumer que les opérateurs `*` et `+` de la grammaire correspondent aux opérateurs arithmétiques standards, car on ne spécifie pas le type des données manipulées. Vous ne pouvez donc pas supposer qu'ils sont commutatifs et associatifs. Par exemple, bien qu'on ait* $(x * y) * z = x * y * z$ *, on n'a pas* $x * (y * z) = x * y * z$ *(par exemple, il pourrait s'agir de nombres à point flottant – avec la norme IEEE 754 sur 32 bits, par exemple, avec les valeurs* $x = 16777216.0$*,* $y = 1.0$*,* $z = 1.0$*,* $x + (y + z)$ *donne* 16777218.0*, et* $(x + y) + z$ *donne* 16777216.0*) et on n'a pas* $x * y = y * x$ *(par exemple si* x *et* y *sont des matrices).*

— **Exercice 5.3.3**. (Exercice 5.5 dans la 1ère édition.)

— **Exercices 5.4.2** et **5.4.3**. (Similaire à l'exercice 5.11 dans la 1ère édition mais seulement en référence à l'exercice 5.6(a).)

— **Exercice 5.5.6**, première directive seulement. (Similaire à l'exercice 5.15 dans la 1ère édition.)

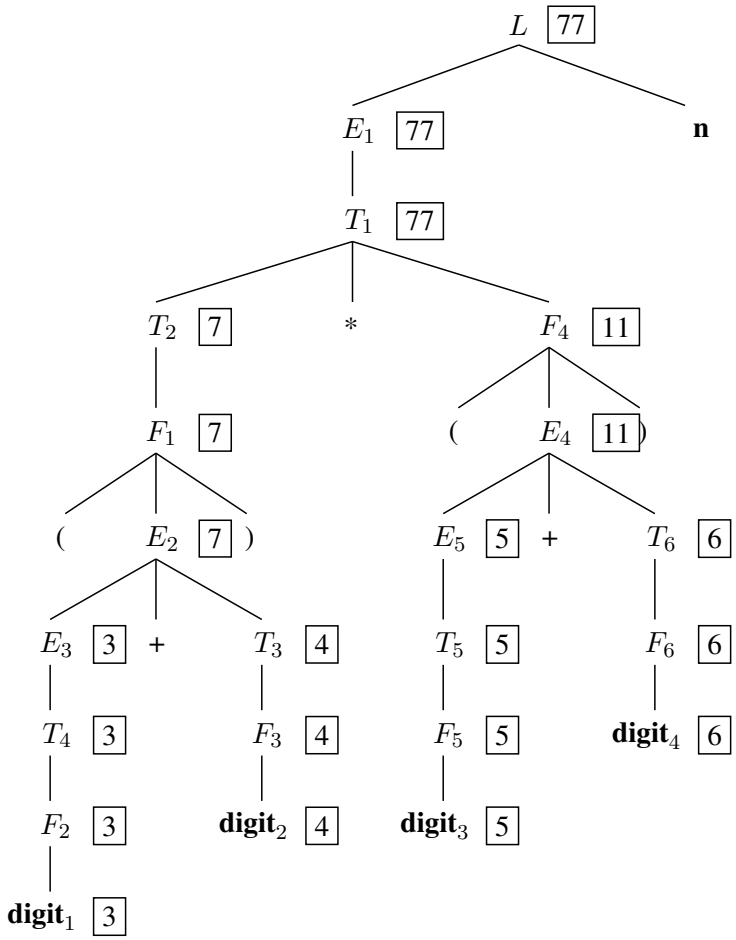
— **Exercice supplémentaire 1**. En utilisant la grammaire trouvée au numéro 5.2.4 et l'entrée `1.01`,
  — i) donnez l'arbre de dérivation
  — ii) tracez un graphe des dépendances
  — iii) fournissez un tri topologique du graphe des dépendances.
— **Exercice supplémentaire 2**. En utilisant la définition orientée-syntaxe de la figure 5.10, donnez l'arbre de syntaxe annoté ainsi que l'arbre de syntaxe abstraite généré pour l'entrée `x - (6 - y) + 5`.
— **Exercice supplémentaire 3**. Ajoutez des règles sémantiques à la grammaire suivante de façon à générer un arbre de syntaxe abstraite :

$$
\begin{aligned}
\textit{bexpr} &\rightarrow \textit{bexpr}\ \textbf{or}\ \textit{bterm} \mid \textit{bterm} \\
\textit{bterm} &\rightarrow \textit{bterm}\ \textbf{and}\ \textit{bfac} \mid \textit{bfac} \\
\textit{bfac} &\rightarrow \textbf{not}\ \textit{bfac} \mid (\textit{bexpr}) \mid \textbf{true} \mid \textbf{false}
\end{aligned}
$$

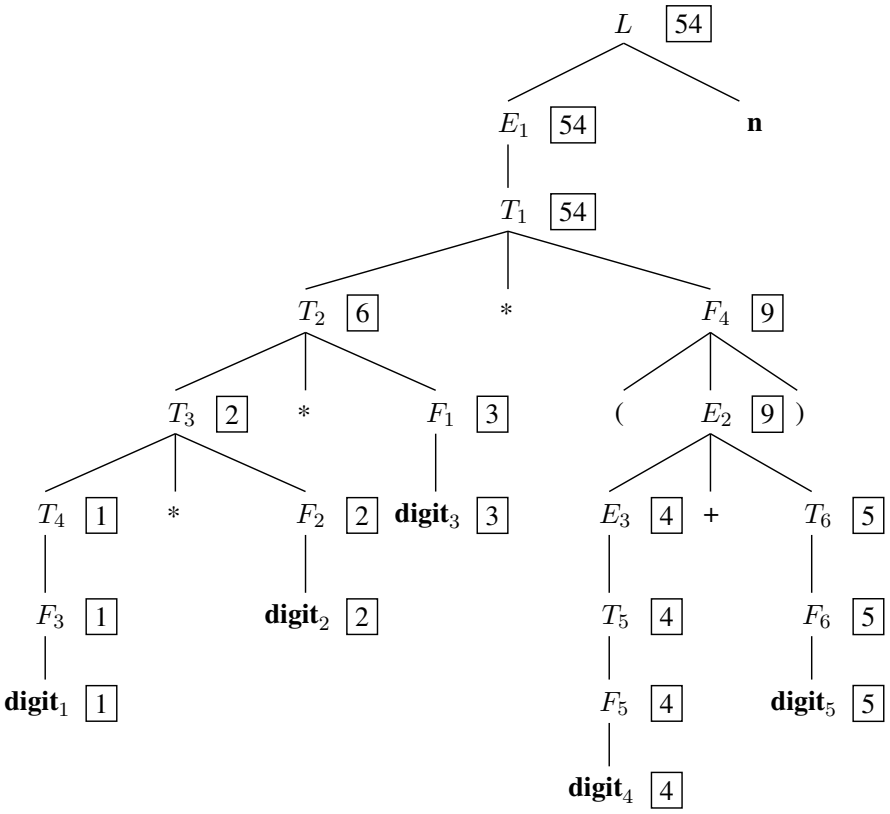
  Les types de noeuds de l'arbre de syntaxe abstraite sont les suivants :
  — **and** : prend deux arguments, lesquels sont les noeuds correspondant aux opérandes
  — **or** : prend deux arguments, lesquels sont les noeuds correspondant aux opérandes
  — **not** : prend un argument, lequel est le noeud correspondant à l'opérande
  — **cb** (*constante booléene*) : prend un argument, lequel est la valeur de la constante booléene (**true** ou **false**).
— **Exercice supplémentaire 4**. Éliminez les récursions à gauche du système de traduction de la figure 5.18.

5.1.1

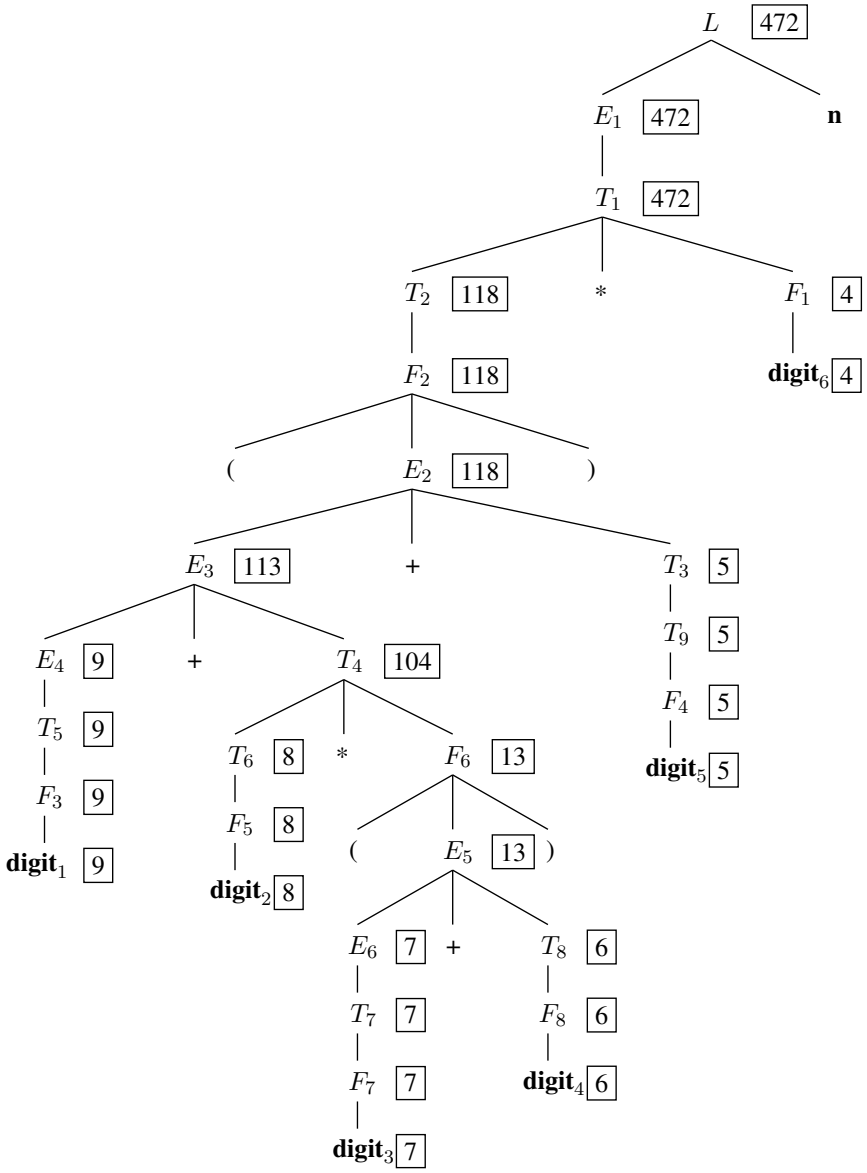
— (a)


L 77
E1 77
n
T1 77
T2 7
*
F4 11
F1 7
(E4 11)
(E2 7)
E5 5 + T6 6
E3 3 + T3 4
T5 5
F6 6
T4 3
F3 4
F5 5
digit4 6
F2 3
digit2 4
digit3 5
digit1 3

— (b)


L 54
E1 54
n
T1 54
T2 6
*
F4 9
T3 2
*
F1 3
(E2 9)
T4 1
*
F2 2
digit3 3
E3 4
+
T6 5
F3 1
digit2 2
T5 4
F6 5
digit1 1
F5 4
digit5 5
digit4 4

— (c)

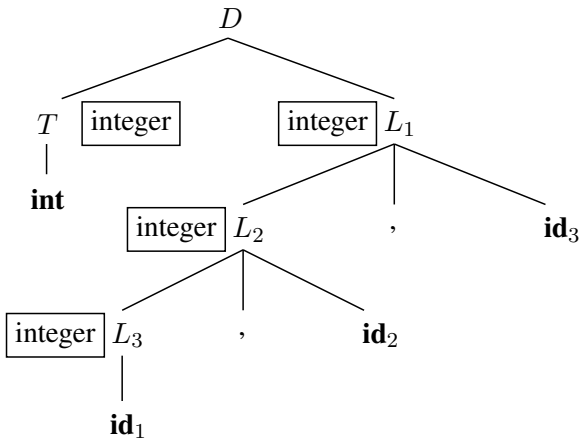

L 472
E1 472
n
T1 472
T2 118
*
F1 4
F2 118
digit6 4
(E2 118)
E3 113
+
T3 5
T9 5
F4 5
digit5 5
E4 9
+
T4 104
T5 9
F3 9
digit1 9
T6 8
*
F6 13
F5 8
digit2 8
(E5 13)
E6 7
+
T8 6
T7 7
F8 6
F7 7
digit4 6
digit3 7

5.2.1

1, 2, 3, 4, 5, 6, 7, 8, 9
1, 2, 3, 5, 4, 6, 7, 8, 9
1, 2, 4, 3, 5, 6, 7, 8, 9
1, 3, 2, 4, 5, 6, 7, 8, 9
1, 3, 2, 5, 4, 6, 7, 8, 9
1, 3, 5, 2, 4, 6, 7, 8, 9
2, 1, 3, 4, 5, 6, 7, 8, 9
2, 1, 3, 5, 4, 6, 7, 8, 9
2, 1, 4, 3, 5, 6, 7, 8, 9
2, 4, 1, 3, 5, 6, 7, 8, 9

5.2.2

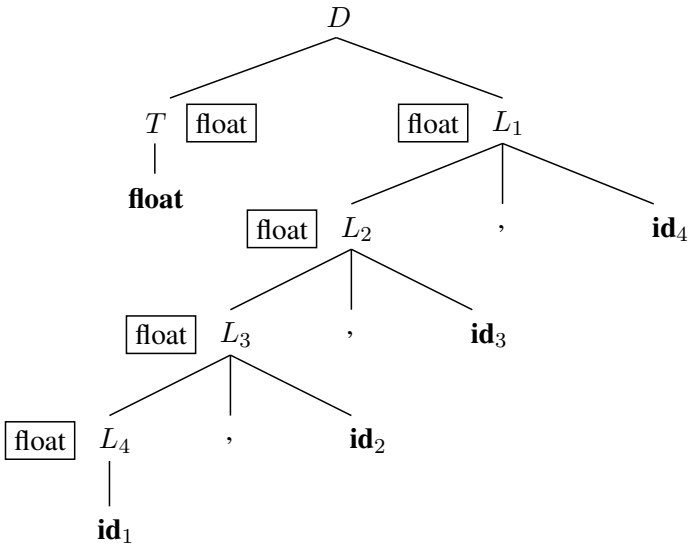
— (a)



Et on obtient la table de symboles suivante :

Symbole	**Nom**	**Type**
$\mathbf{id}_1$	a	integer
$\mathbf{id}_2$	b	integer
$\mathbf{id}_3$	c	integer

— (b)



Et on obtient la table de symboles suivante :

Symbole	**Nom**	**Type**
$\mathbf{id}_1$	w	float
$\mathbf{id}_2$	x	float
$\mathbf{id}_3$	y	float
$\mathbf{id}_4$	z	float

5.2.3

— (a)
  — (i). Non, car il y a utilisation d'un attribut hérité.
  — (ii). Oui
  — (iii). Oui
— (b)
  — (i). Non, car il y a utilisation d'un attribut hérité.
  — (ii). Oui
  — (iii). Oui
— (c)
  — (i). Oui
  — (ii). Oui
  — (iii). Oui
— (d)
  — (i). Non, car il y a utilisation d'attributs hérités.
  — (ii). Non, car l'attribut hérité de B dépend de C, qui est à sa droite.
  — (iii). Non, car il y a une boucle de dépendances : $D.i$ dépend de $B.i$, qui dépend de $A.s$, qui dépend de $D.i$.

5.2.4

S	$\rightarrow L_1 \ . \ L_2$	$L_1.side := \text{left}$
		$L_2.side := \text{right}$
		$S.val := L_1.v + L_2.v$
S	$\rightarrow L$	$L.side := \text{left}$
		$S.val := L.v$
L	$\rightarrow L_1 B$	$L_1.side := L.side$
		$L.c := \frac{L_1.c}{2}$
		$L.v := (L.side = \text{left}) \ ? \ (2 * L_1.v + B.v) \ : \ (L_1.v + B.v * L.c)$
L	$\rightarrow B$	$L.c := 0.5$
		$L.v := (L.side = \text{left}) \ ? \ (B.v) \ : \ (B.v * L.c)$
B	$\rightarrow$ `0`	$B.v := 0$
B	$\rightarrow$ `1`	$B.v := 1$

Fonctionnement de cette grammaire :

— l'attribut *side* est hérité par L puis par B pour permettre de savoir si le symbole se trouve dans la partie entière ou fractionnaire du nombre ;

— Pour la partie fractionnaire : prenons par exemple `0.101`, qui est équivalent à

$$1 * 2^{-1} \ + \ 0 * 2^{-2} \ + \ 1 * 2^{-3}$$

Les puissances de 2 qui apparaissent ici sont représentées par l'attribut c ("contribution") dans la grammaire. Le premier bit après le point reçoit une contribution de 2^{-1} (0.5), le second bit après le point, une contribution de 2^{-2} (0.25), etc. L'attribut c est initialisé à 0.5 pour le premier bit, puis est divisé par deux pour chaque bit successif.

— Pour la partie entière : on se contente de multiplier par deux ce qui vient avant le bit courant, puis d'additionner le nouveau bit. Par exemple, pour `1011`, on procéderait ainsi :

- Lire le bit `1` ; mettre la valeur à 1
- Multiplier la valeur courante par 2 (on obtient 2) et additionner le bit courant (0)
- Multiplier la valeur courante par 2 (on obtient 4) et additionner le bit courant (1), pour obtenir 5
- Multiplier la valeur courante par 2 (on obtient 10) et additionner le bit courant (1), pour obtenir 11.

5.2.5

Une grammaire S-attribuée ne peut contenir que des attributs synthétisés, nous devons donc retirer tous les attributs hérités. Une façon de procéder est de calculer deux valeurs pour chaque L, une pour le cas où ce L est dans la portion entière et une pour le cas où ce nombre est dans la section décimale. S peut alors choisir l'attribut pertinent.

$S \rightarrow L_1 \ . \ L_2$	$S.val := L_1.intval \ + \ L_2.decval$
$S \rightarrow L$	$S.val := L.intval$
$L \rightarrow L_1 \ B$	$L.intval := L_1.intval * 2 \ + \ B.val$
	$L.depth := L_1.depth \ + \ 1$
	$L.decval := L_1.decval \ + \ 2^{-L.depth} * B.val$
$L \rightarrow B$	$L.intval := B.val$
	$L.decval := B.val * 2^{-1}$
	$L.depth := 1$
$B \rightarrow$ `0`	$B.val := 0$
$B \rightarrow$ `1`	$B.val := 1$

5.2.6

Notez que l'énoncé demande une grammaire *appropriée pour l'analyse descendante* ; nous devons donc produire une grammaire sans récursion à gauche, sans préfixe commun, sans ambiguité, etc.

Utilisons la grammaire suivante pour reconnaitre les expressions régulières :

$$
\begin{array}{rcl}
S & \rightarrow & T \ S' \\
S' & \rightarrow & \textbf{barre} \ T \ S' \ \mid \epsilon \\
T & \rightarrow & R \ T' \\
T' & \rightarrow & R \ T' \ \mid \epsilon \\
R & \rightarrow & B \ R' \\
R' & \rightarrow & *R' \ \mid \epsilon \\
B & \rightarrow & \textbf{char} \ \mid \ (\ S\ ) \ \mid \ \textbf{Epsilon}
\end{array}
$$

Notez que les jetons correspondant à la barre verticale sont ici notés **barre** afin d'éviter toute confusion avec la barre verticale communément utilisée pour séparer deux productions d'une grammaire ; de même, le jeton **Epsilon** sera utilisé pour dénoter un ϵ apparaissant dans une expression régulière.

Puisque tous les graphes générés par l'algorithme n'ont qu'un état final, nous représenterons un graphe comme un couple de noeuds, où le premier élément du couple est l'état initial et le second, l'état final.

Nous obtenons la définition orientée syntaxe suivante :

S	$\rightarrow$	TS'	$S'.i$	$:=$	$T.s$
			$S.s$	$:=$	$S'.s$
S'	$\rightarrow$	**barre** TS'_1	$S'_1.i$	$:=$	$f\_ou(S'.i,\ T.s)$
			$S'.s$	$:=$	$S'_1.s$
S'	$\rightarrow$	ϵ	$S'.s$	$:=$	$S'.i$
T	$\rightarrow$	RT'	$T'.i$	$:=$	$R.s$
			$T.s$	$:=$	$T'.s$
T'	$\rightarrow$	RT'_1	$T'_1.i$	$:=$	$f\_concat(T'.i,\ R.s)$
			$T'.s$	$:=$	$T'_1.s$
T'	$\rightarrow$	ϵ	$T'.s$	$:=$	$T'.i$
R	$\rightarrow$	BR'	$R'.i$	$:=$	$B.s$
			$R.s$	$:=$	$R'.s$
R'	$\rightarrow$	$*R'_1$	$R'_1.i$	$:=$	$f\_ferm(R'.i)$
			$R'.s$	$:=$	$R'_1.s$
R'	$\rightarrow$	ϵ	$R'.s$	$:=$	$R'.i$
B	$\rightarrow$	**char**	$B.s$	$:=$	$f\_lire\_char($ **char** $)$
B	$\rightarrow$	$(\ S\ )$	$B.s$	$:=$	$S.s$
B	$\rightarrow$	ϵ	$B.s$	$:=$	$f\_lire\_epsilon()$

Cette définition utilise les fonctions suivantes (en pseudo-code, où nous supposons que la fonction `ajouter_arc` est définie et ajoute un arc au graphe, sans nous soucier des détails de son implémentation) :

```
/** Correspond à la figure 3.40 du manuel
  * Reçoit en argument deux couples représentant deux automates
  * Retourne un couple représentant un automate qui reconnait
  * l'union dans langages reconnus par les deux automates reçus
  */
func f_ou((i1, f1), (i2, f2))
{
    i := new Node();
    f := new Node();
    ajouter_arc(i, ϵ, i1);
    ajouter_arc(i, ϵ, i2);
    ajouter_arc(f1, ϵ, f);
    ajouter_arc(f2, ϵ, f);
    return (i, f);
}
```

```
/** Reçoit en argument deux couples représentant deux automates
  * Retourne un couple représentant un automate qui reconnait la
  * concaténation dans langages reconnus par les deux automates reçus
  */
func f_concat((i_1, f_1), (i_2, f_2))
{
    ajouter_arc(f_1, ϵ, i_2);
    return (i_1, f_2);
}

/** Correspond à la figure 3.42 du manuel
  * Reçoit en argument un couple représentant un automate
  * Retourne un couple représentant un automate qui reconnait
  * la fermeture du langage de l'automate reçu en argument
  */
func f_ferm((i_1, f_1))
{
    i := new Node();
    f := new Node();
    ajouter_arc(i, ϵ, i_1);
    ajouter_arc(i, ϵ, f);
    ajouter_arc(f_1, ϵ, i_1);
    ajouter_arc(f_1, ϵ, f);
    return (i, f);
}

/** Reçoit un caractère, retourne un automate qui lit ce caractère */
func f_lire_char(char c)
{
    i := new Node();
    f := new Node();
    ajouter_arc(i, c, f);
    return (i, f);
}

/** Retourne un automate qui lit epsilon */
func f_lire_epsilon()
{
    i := new Node();
    f := new Node();
    ajouter_arc(i, ϵ, f);
    return (i, f);
}
```

5.3.1

— (a)

$E \rightarrow E_1 + T$	$E.type :=$	**if** $E_1.type =$ `float` **or** $T.type =$ `float` **then** `float` **else** `int`
$E \rightarrow T$	$E.type :=$	$T.type$
$T \rightarrow$ **num** . **num**	$T.type :=$	`float`
$T \rightarrow$ **num**	$T.type :=$	`integer`

— (b) Ajoutons l'attribut `pf`, un chaîne de caractères contenant l'expression en notation postfixe.

$E \rightarrow E_1 + T$	$E.type :=$	**if** $E_1.type =$ `float` **or** $T.type =$ `float` **then** `float` **else** `int`
	$E.pf :=$	$E_1.pf \parallel T.pf \parallel$ ' + '
$E \rightarrow T$	$E.type :=$	$T.type$
	$E.pf :=$	$T.pf$
$T \rightarrow$ **num**$_1$. **num**$_2$	$T.type :=$	`float`
	$T.pf :=$	**num**$_1$.*text* $\parallel$ '.' $\parallel$ **num**$_2$.*text*
$T \rightarrow$ **num**	$T.type :=$	`integer`
	$T.pf :=$	**num**.*text*

Notez l'utilisation de **num**.*text* *au lieu de* **num**.*val* *(nous supposons ici que "text" contient le texte intégral du jeton, alors que "val" contient sa valeur numérique) ; il est nécessaire d'utiliser "text" et non "val", car pour une chaine comme* `1.001`*, on aurait* $T \rightarrow \textbf{num}_1$. $\textbf{num}_2$ *, avec* $\textbf{num}_2 = 001$ *; si on lisait* $\textbf{num}_2.val$ *, on obtiendrait* 1*, les zéros à gauche ayant été ignorés, ce qui modifierait la valeur de notre nombre à virgule. En lisant* $\textbf{num}_2.text$ *, on s'assure donc de conserver les zéros à gauche.*

5.3.2

La première étape consiste à construire une grammaire qui permet de reconnaître les chaînes du langage tout en respectant la priorité des opérateurs et l'associativité.

$$
\begin{array}{lcl}
S & \rightarrow & S \texttt{ + } T \mid T \\
T & \rightarrow & T \texttt{ * } U \mid U \\
U & \rightarrow & \textbf{id} \mid \texttt{(} S \texttt{)}
\end{array}
$$

On définit l'attribut *pr* qui contient la priorité de l'opérateur "central" d'une expression ; une expression indivisible (identificateur) a $pr = 1$; une expression avec '`*`' comme opérateur principal a $pr = 2$; enfin, une expression avec '`+`' comme opérateur principal a $pr = 3$.

Puisque les opérateurs sont associatifs à gauche, l'opérande gauche de chaque opérateur peut être de niveau *pr* égal à l'opérateur de la production courante sans être parenthésé. Toutefois, l'opérande droit d'un opérateur doit être parenthésé s'il est de niveau *pr* égal ou supérieur à celui de l'opérateur de la production courante afin de respecter la priorité et l'associativité à gauche.

On construit ensuite le système de traduction :

$S \rightarrow S_1 + T$	$S.pr$	:=	3
	$S.str$	:=	$S_1.str \parallel$ '+' $\parallel$
			(**if** $T.pr = 3$ **then** '(' $\parallel$ $T.str$ $\parallel$ ')'
			else $T.str$)
$S \rightarrow T$	$S.pr$	:=	$T.pr$
	$S.str$	:=	$T.str$
$T \rightarrow T_1 * U$	$T.pr$	:=	2
	$T.str$	:=	(**if** $T_1.pr > 2$ **then** '(' $\parallel$ $T_1.str$ $\parallel$ ')'
			else $T_1.str$) $\parallel$ '*' $\parallel$
			(**if** $U.pr >= 2$ **then** '(' $\parallel$ $U.str$ $\parallel$ ')'
			else $U.str$)
$T \rightarrow U$	$T.pr$	:=	$U.pr$
	$T.str$	:=	$U.str$
$U \rightarrow$ **id**	$U.pr$	:=	1
	$U.str$	:=	**id** $.name$
$U \rightarrow (S)$	$U.pr$	:=	$S.pr$
	$U.str$	:=	$S.str$

5.3.3

La première étape consiste à construire une grammaire qui permet de reconnaître les chaînes du langage tout en respectant la priorité des opérateurs.

$$
\begin{array}{lcl}
S & \rightarrow & S + T \mid T \\
T & \rightarrow & T * U \mid U \\
U & \rightarrow & \textbf{id} \mid (\ S\ )
\end{array}
$$

Soient les attributs *org* et *dx*, qui sont respectivement la formule originale (non dérivée) et la formule dérivée.

On construit ensuite le système de traduction :

$$
\begin{array}{lll}
S \rightarrow S_1 \ + \ T & S.org & := \ S_1.org \parallel \text{`+'} \parallel T.org \\
 & S.dx & := \ S_1.dx \parallel \text{`+'} \parallel T.dx \\
S \rightarrow T & S.org & := \ T.org \\
 & S.dx & := \ T.dx \\
T \rightarrow T_1 \ * \ U & T.org & := \ \text{`(`} \parallel T_1.org \parallel \text{`)`} \parallel \text{`*`} \parallel \text{`(`} \parallel U.org \parallel \text{`)`} \\
 & T.dx & := \ \text{`(`} \parallel T_1.org \parallel \text{`)`} \parallel \text{`*`} \parallel \text{`(`} \parallel U.dx \parallel \text{`)`} + \\
 & & \quad \text{`(`} \parallel T_1.dx \parallel \text{`)`} \parallel \text{`*`} \parallel \text{`(`} \parallel U.org \parallel \text{`)`} \\
T \rightarrow U & T.org & := \ U.org \\
 & T.dx & := \ U.dx \\
U \rightarrow \mathbf{x} & U.org & := \ \text{`}x\text{'} \\
 & U.dx & := \ \text{`1'} \\
U \rightarrow (\ S\ ) & U.org & := \ S.org \\
 & U.dx & := \ S.dx
\end{array}
$$

5.4.2

$$
\begin{array}{rcl}
A & \rightarrow & \mathtt{0}A' \\
A' & \rightarrow & \{a\}B\ A' \ \mid \ B\ \{b\}A' \ \mid \ \epsilon \\
B & \rightarrow & \mathtt{1}B' \\
B' & \rightarrow & \{c\}A\ B' \ \mid \ A\ \{d\}B' \ \mid \ \epsilon
\end{array}
$$

5.4.3

$$
\begin{array}{rcl}
B & \rightarrow & 1\ \{B'.i := 1\}\ B'\ \{B.val := B'.s\} \\
B' & \rightarrow & 0\ \{B'_1.i := 2 * B'.i\}\ B'_1\ \{B'.s := B'_1.s\} \\
B' & \rightarrow & 1\ \{B'_1.i := 2 * B'.i + 1\}\ B'_1\ \{B'.s := B'_1.s\} \\
B' & \rightarrow & \{B'.s := B'.i\}
\end{array}
$$

5.5.6

Premièrement, notons que la grammaire donnée plus haut pour l'exercice 5.2.4 contenait des récursions à gauche. Commençons donc par produire une version modifiée de la grammaire qui ne comporte pas de récursions à gauche. Un attribut supplémentaire *acc* a été ajouté pour contenir la valeur des nombres entiers ; cet attribut hérité est ensuite "remonté" vers S par l'attribut synthéthisé v.

$$
\begin{array}{lcl}
S & \rightarrow & \{L_1.side := \text{left}\}\ L_1\ \textbf{.}\ \{L_2.side := \text{right}\}\ L_2\ \{S.val := L_1.v + L_2.v\} \\
S & \rightarrow & \{L.side := \text{left}\}\ L\ \{S.val := L.v\} \\
L & \rightarrow & B\ \{R.c := 0.5;\ R.side := L.side\}\ R \\
 & & \{L.v := (L.side = \text{left})\ ?\ R.v\ :\ (R.v + B.v * R.c)\} \\
R & \rightarrow & B\ \{R_1.side := R.side;\ R_1.c := \frac{R.c}{2};\ R_1.acc := R.acc * 2 + B.v\ \}\ R_1 \\
 & & \{R.v := (R.side = \text{left})\ ?\ R_1.v\ :\ (R_1.v + B.v * R_1.c)\} \\
R & \rightarrow & \epsilon\ \{R.v := (R.side = \text{left})\ ?\ R.acc\ :\ 0\} \\
B & \rightarrow & \textbf{0}\ \{B.v := 0\} \\
B & \rightarrow & \textbf{1}\ \{B.v := 1\}
\end{array}
$$

Pour distinguer les deux productions de S, il suffira d'observer si on trouve un point après avoir lu un premier L.

Distinguer les deux productions de R est moins évident. Il faudra commencer par consommer un B, puis observer le prochain jeton. Si le prochain jeton est un `0` ou un `1`, nous devons générer un B supplémentaire afin de lire ce caractère, donc il s'agit de la première production de R.

On obtient alors le code suivant (ici en utilisant la syntaxe de python 2.7) :

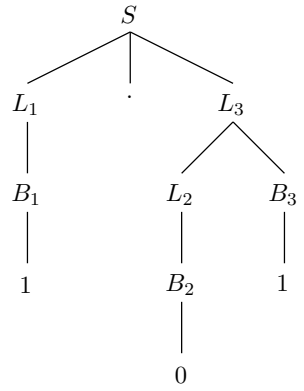
```
# Fonction pour lire un symbole 'B'
def B(fileJetons):
    if len(fileJetons) < 1 :
        raise Exception("Erreur de syntaxe (le mot finit trop vite)")

    if fileJetons[0] == '0':
        fileJetons.popleft()  # consommer le caractère
        return 0.0            # B.v = 0
    if fileJetons[0] == '1':
        fileJetons.popleft()  # consommer le caractère
        return 1.0            # B.v = 1

    # ni 0 ni 1
    raise Exception("Erreur de syntaxe; à la place de '" + fileJetons[0] +
                    "', il faudrait 0 ou 1")
```

```
# Fonction pour lire un symbole 'R'
def R(fileJetons, acc, side, c):

    if (len(fileJetons) < 1 or fileJetons[0] not in ['0', '1']):
        # Production R -> Epsilon
        if (side == 'left'):
            R_v = acc
        else:
            R_v = 0
    else:
        # Production R -> B R
        B_v = B(fileJetons)

        if (side == 'left'):
            # L'attribut hérité 'c' n'est pas utilisé quand side == 'left'
            # donc on évite de le calculer inutilement
            R1_v = R(fileJetons, acc*2 + B_v, side, None)
            R_v = R1_v
        else:
            # L'attribut hérité 'acc' n'est pas utilisé quand side == 'right'
            # donc on évite de le calculer inutilement
            R1_c = c / 2
            R1_v = R(fileJetons, None, side, R1_c)
            R_v = R1_v + B_v*R1_c

    return R_v

# Fonction pour lire un symbole 'L'
def L(fileJetons, side):
    B_v = B(fileJetons)
    R_c = 0.5
    R_v = R(fileJetons, B_v, side, R_c)

    if (side == 'left'):
        L_v = R_v
    else:
        L_v = R_v + B_v * R_c

    return L_v

# Fonction pour lire un symbole 'S'
def S(fileJetons):
    L1_val = L(fileJetons, 'left')

    if (len(fileJetons) > 0 and fileJetons[0] == '.'):
        # Première production (S -> L.L)
        fileJetons.popleft() # consommer le point
        L2_val = L(fileJetons, 'right')
        return L1_val + L2_val
    else:
        # Seconde production (S -> L)
        return L1_val
```

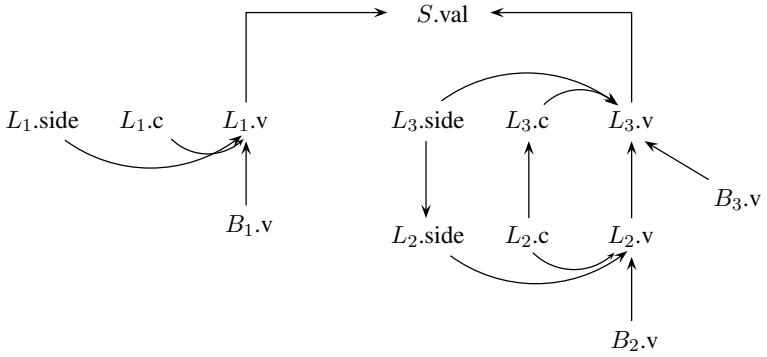
```
# Fonction principale pour lire un mot de la chaine
def parse(fileJetons):
    val = S(fileJetons)

    # Vérifier que tout le mot a été lu
    if len(fileJetons) > 0:
        raise Exception("Erreur de syntaxe, fin pas atteinte")

    return val

# Exemple d'utilisation
import collections
v = parse( collections.deque(['1','0','1','.','1','0','1']) )
print "101.101 : ", v
```

Exercice Supplémentaire 1

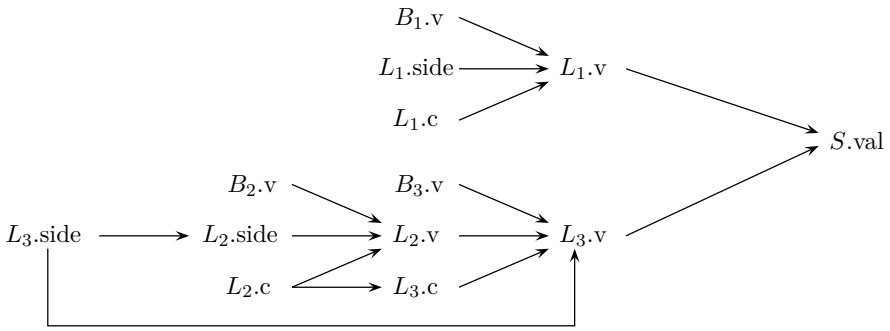
— i)



— 2)

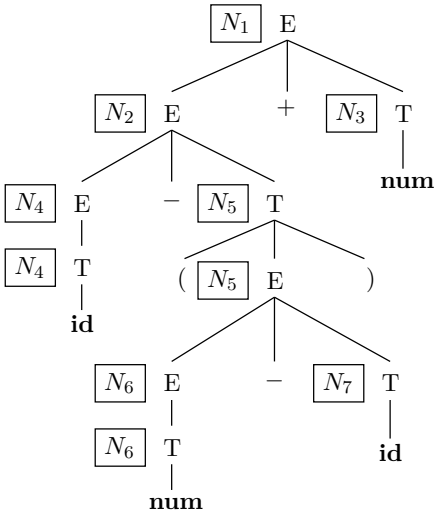


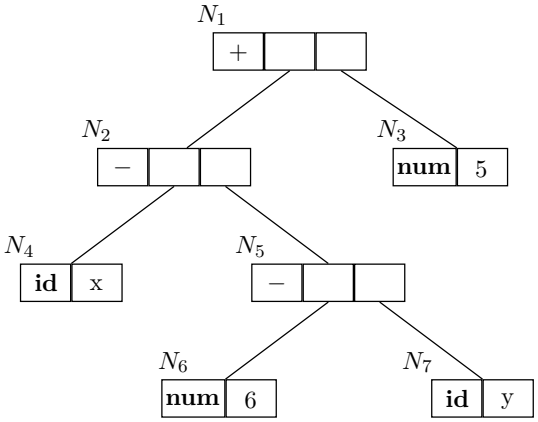
— 3)



À partir de ce graphe trié topologiquement, on peut aisément trouver un ordre d'évaluation. Un exemple d'ordre d'évaluation serait :
$B_1.v$, $L_1.side$, $L_1.c$, $B_2.v$, $B_3.v$, $L_3.side$, $L_2.side$, $L_2.c$, $L_3.c$, $L_1.v$, $L_2.v$, $L_3.v$, $S.val$

Exercice Supplémentaire 2





Exercice Supplémentaire 3

Productions	Règles Sémantiques
$bexpr \rightarrow bexpr_1$ **or** $bterm$	$bexpr.node :=$ **new** Node(**or**, $bexpr_1.node$, $bterm.node$)
$bexpr \rightarrow bterm$	$bexpr.node := bterm.node$
$bterm \rightarrow bterm_1$ **and** $bfac$	$bterm.node :=$ **new** Node(**and**, $bterm_1.node$, $bfac.node$)
$bterm \rightarrow bfac$	$bterm.node := bfac.node$
$bfac \rightarrow$ **not** $bfac_1$	$bfac.node :=$ **new** Node(**not**, $bfac_1.node$)
$bfac \rightarrow$ ($bexpr$)	$bfac.node := bexpr.node$
$bfac \rightarrow$ **true**	$bfac.node :=$ **new** Leaf(**cb**, **true**)
$bfac \rightarrow$ **false**	$bfac.node :=$ **new** Leaf(**cb**, **false**)

Exercice Supplémentaire 4

$$
\begin{array}{rcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val)\ \} \\
E & \rightarrow & T & \{\ E'.i := T.val\ \} \\
 & & E' & \{\ E.val := E'.val\ \} \\
E' & \rightarrow & +\ T & \{\ E'_1.i := E'.i + T.val\ \} \\
 & & E'_1 & \{\ E'.val := E'_1.val\ \} \\
E' & \rightarrow & \epsilon & \{\ E'.val := E'.i\ \} \\
T & \rightarrow & F & \{\ T'.i := F.val\ \} \\
 & & T' & \{\ T.val := T'.val\ \} \\
T' & \rightarrow & *\ F & \{\ T'_1.i := T'.i * F.val\ \} \\
 & & T'_1 & \{\ T'.val := T'_1.val\ \} \\
T' & \rightarrow & \epsilon & \{\ T'.val := T'.i\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val := E.val\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val := \mathbf{digit}.lexval\ \}
\end{array}
$$