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

Danny Dubé
Automne 2018
Version: 10 décembre

Corrigé du travail pratique #2

Réponses

Note: les réponses aux différentes questions ne sont pas toujours uniques.

1. Dans les réponses aux sous-questions (a) à (c), c'est la première grammaire hors-contexte que nous avons adoptée, tandis que dans celles pour (d) et (e), c'est la seconde.

   (a) On ajoute au non-terminal A quatre attributs synthétisés: a, b, c et ok. Soit w la chaîne d'entrée. Considérons l'arbre de dérivation pour w. Soit A_i un des noeud de l'arbre et w_i le suffixe de w décrit par le sous-arbre enraciné en A_i. La chaîne d'entrée est analysée de droite à gauche au fur et à mesure que les calculs faits par la définition orientée-syntaxe (DOS) remontent le long de la colonne vertébrale de l'arbre. L'attribut $A_i.a$ indique la longueur de la répétition de '`a`' qui débute w_i. Les attributs $A_i.b$ et $A_i.c$ ont des significations similaires. Notons qu'on a nécessairement au moins deux des attributs $A_i.a$, $A_i.b$ et $A_i.c$ qui sont nuls. L'attribut $A_i.ok$ indique si w_i respecte la restriction sur les longueurs des répétitions.

Productions	Règles sémantiques
$S \to A$	$S.ok := A.ok$
$A \to \texttt{a}\, A_1$	$A.a := 1 + A_1.a$ $A.b := 0$ $A.c := 0$ $A.ok := (A.a < 2018) \wedge A_1.ok$
$A \to \texttt{b}\, A_1$	$A.a := 0$ $A.b := 1 + A_1.b$ $A.c := 0$ $A.ok := (A.b < 2018) \wedge A_1.ok$
$A \to \texttt{c}\, A_1$	$A.a := 0$ $A.b := 0$ $A.c := 1 + A_1.c$ $A.ok := (A.c < 2018) \wedge A_1.ok$
$A \to \epsilon$	$A.a := 0$ $A.b := 0$ $A.c := 0$ $A.ok :=$ vrai

(b) On ajoute au non-terminal A deux attributs synthétisés: l et n. À nouveau, soit w_i le suffixe de la chaîne d'entrée décrit par le sous-arbre enraciné en A_i. Notre attribut $A_i.l$ indique la longueur du plus long suffixe de `baca` qui débute w_i; i.e., $A_i.l = 4$ si w_i commence par `baca`, sinon $A_i.l = 3$ si w_i commence par `aca`, sinon $A_i.l = 2$ si w_i commence par `ca`, sinon ... L'attribut $A.n$ indique le nombre de fois où la sous-chaîne `baca` apparaît dans w_i. L'essentiel de la difficulté dans la DOS consiste en le calcul de l'attribut $A_i.l$.

Productions	Règles sémantiques			
$S \to A$	$S.ok$	$:=$	$(A.n = 2018)$	
$A \to \texttt{a}\, A_1$	$A.l$	$:=$	**if** $A_1.l = 2$ **then** 3 **else** 1	// $\texttt{a} \cdot \texttt{ca}$?
	$A.n$	$:=$	$A_1.n$	
$A \to \texttt{b}\, A_1$	$A.l$	$:=$	**if** $A_1.l = 3$ **then** 4 **else** 0	// $\texttt{b} \cdot \texttt{aca}$?
	$A.n$	$:=$	**if** $A.l = 4$ **then** $1 + A_1.n$ **else** $A_1.n$	
$A \to \texttt{c}\, A_1$	$A.l$	$:=$	**if** $A_1.l \in \{1, 3\}$ **then** 2 **else** 0	// $\texttt{c} \cdot \texttt{a}[\texttt{ca}]$?
	$A.n$	$:=$	$A_1.n$	
$A \to \epsilon$	$A.l$	$:=$	0	
	$A.n$	$:=$	0	

(c) On ajoute au non-terminal A trois attributs synthétisés: abc, bc et c. Encore, on suppose que w_i est écrit sous le noeud A_i. Les trois attributs $A_i.abc$, $A_i.bc$ et $A_i.c$ indiquent de combien de façons on peut extraire les sous-séquences `abc`, `bc` et `c` de w_i, respectivement. La logique du comptage est la suivante, lorsqu'on se trouve à la position d'un '`a`', i.e. lorsque $w_i = \texttt{a} \cdot w_{i+1}$:

- il y a le même nombre de façons d'extraire `c` de w_i que de w_{i+1};
- il y a le même nombre de façons d'extraire `bc` de w_i que de w_{i+1};
- cependant, pour compter le nombre de façons d'extraire `abc` de w_i, il faut additionner les nombres de façons d'extraire `bc` et `abc` de w_{i+1}, car on a le droit de prendre le '`a`' courant ou pas.

La logique du comptage est similaire, dans les autres cas.

Productions	Règles sémantiques
$S \to A$	$S.ok := A.abc \geq 2018$
$A \to \texttt{a}\, A_1$	$A.abc := A_1.bc + A_1.abc$
	$A.bc := A_1.bc$
	$A.c := A_1.c$
$A \to \texttt{b}\, A_1$	$A.abc := A_1.abc$
	$A.bc := A_1.c + A_1.bc$
	$A.c := A_1.c$
$A \to \texttt{c}\, A_1$	$A.abc := A_1.abc$
	$A.bc := A_1.bc$
	$A.c := 1 + A_1.c$
$A \to \epsilon$	$A.abc := 0$
	$A.bc := 0$
	$A.c := 0$

(d) On ajoute aux non-terminaux A, B et C certains des trois attributs synthétisés a, b et c, selon les besoins. Ceux-ci comptent les nombres d'apparitions des terminaux des mêmes noms, respectivement.

Productions	Règles sémantiques		
$S \to A$	$S.ok := (2 \cdot A.a = A.b) \vee (3 \cdot A.b = A.c) \vee (4 \cdot A.c = A.a)$		
$A \to \texttt{a}\, A_1$	$A.a := 1 + A_1.a;$	$A.b := A_1.b;$	$A.c := A_1.c$
$A \to B$	$A.a := 0;$	$A.b := B.b;$	$A.c := B.c$
$B \to \texttt{b}\, B_1$		$B.b := 1 + B_1.b;$	$B.c := B_1.c$
$B \to C$		$B.b := 0;$	$B.c := C.c$
$C \to \texttt{c}\, C_1$			$C.c := 1 + C_1.c$
$C \to \epsilon$			$C.c := 0$

(e) La réponse est presqu'identique à celle en (d).

Productions	Règles sémantiques		
$S \rightarrow A$	$S.ok := A.a + (A.b)^2 = (A.c)^3$		
$A \rightarrow \texttt{a}\, A_1$	$A.a := 1 + A_1.a;$	$A.b := A_1.b;$	$A.c := A_1.c$
$A \rightarrow B$	$A.a := 0;$	$A.b := B.b;$	$A.c := B.c$
$B \rightarrow \texttt{b}\, B_1$		$B.b := 1 + B_1.b;$	$B.c := B_1.c$
$B \rightarrow C$		$B.b := 0;$	$B.c := C.c$
$C \rightarrow \texttt{c}\, C_1$			$C.c := 1 + C_1.c$
$C \rightarrow \epsilon$			$C.c := 0$

2. La stratégie qu'on adopte ici consiste à vérifier si on peut trouver un entier dans la liste tel qu'on observe qu'il y a eu croissance jusqu'à cet entier et qu'il y a décroissance à partir de cet entier. Quand une telle vérification est un succès, on s'assure de faire remonter ce succès en plaçant un booléen vrai dans l'attribut synthétisé $L_i.ok$. Pour vérifier qu'il y a croissance jusqu'à un entier, le noeud L_i qui produit directement cet entier hérite d'un attribut $L_i.h$. Pour arriver à tester la croissance jusqu'à un entier, on hérite aussi de la valeur de l'entier précédent dans l'attribut $L_i.p$. Similairement, pour vérifier qu'il y a décroissance à partir d'un certain entier, le noeud L_i qui produit directement cet entier synthétise un attribut $L_i.b$. Pour arriver à tester la décroissance à partir d'un entier, on synthétise aussi la valeur de l'entier suivant dans l'attribut $L_i.s$.

Productions	Règles sémantiques		
$S \rightarrow$ `[ ]`	$S.ok$	:=	vrai
$S \rightarrow$ `[` **int** L	$L.h$	:=	vrai
	$L.p$	:=	**int**.*lexval*
	$S.ok$	:=	$((\textbf{int}.lexval \geq L.s) \wedge L.b) \vee L.ok$
$L \rightarrow$ `,` **int** L_1	$L_1.h$	:=	$L.h \wedge (L.p \leq \textbf{int}.lexval)$
	$L_1.p$	:=	**int**.*lexval*
	$L.b$	:=	$(\textbf{int}.lexval \geq L_1.s) \wedge L_1.b$
	$L.s$	:=	**int**.*lexval*
	$L.ok$	:=	$(L_1.h \wedge L.b) \vee L_1.ok$
$L \rightarrow$ `]`	$L.b$	:=	vrai
	$L.s$	:=	$-\infty$
	$L.ok$	:=	faux

3. Afin de vérifier que le monceau est semi-ordonné, on fait synthétiser dans $T.val$ la valeur de la clé stockée à la racine de l'arbre générée par T. Afin de vérifier que le monceau est un arbre complet, on mesure la profondeur de tous les sous-arbres vides et on s'assure que la différence de profondeur entre celui qui est au plus bas niveau et celui qui est au plus haut niveau est d'au plus 1. Afin de vérifier que le monceau est un arbre tassé à gauche, on s'assure que la profondeur des sous-arbres vides va en diminuant, de gauche à droite. Ces deux dernières vérifications se font à l'aide de deux attributs, $T.g$ et $T.d$, qui indiquent respectivement la profondeur du sous-arbre vide le plus à gauche et celle du sous-arbre vide le plus à droite. Lorsqu'une violation de la première ou de la troisième propriété est détectée à un noeud T_i, on fixe $T_i.val$ à $+\infty$. Cette valeur garantit de produire une violation de la première propriété chez le parent. Ainsi, ce mécanisme entraîne une propagation de ce signal d'erreur. Une violation de la deuxième propriété n'est détectée qu'au niveau de la racine de l'arbre.

Productions	Règles sémantiques
$S \rightarrow T$	$S.ok := (T.val < +\infty) \wedge (T.g - T.d \leq 1)$
$T \rightarrow$ (T_1 / **num** \ T_2)	$T.val :=$ **if** $(\textbf{num}.lexval > T_1.val) \wedge$ $(\textbf{num}.lexval > T_2.val) \wedge$ $(T_1.d \geq T_2.g)$ **then num**.*lexval* **else** $+\infty$
	$T.g := T_1.g + 1$
	$T.d := T_2.d + 1$
$T \rightarrow$.	$T.val := -\infty$
	$T.g := 0$
	$T.d := 0$

4. Seules les L-productions doivent être modifiées. En effet, la S-production ne comporte pas de récursion à gauche (et elle n'est pas impliquée dans une récursion à gauche indirecte non plus). De plus, le fait d'éliminer la récursion à gauche dans les L-productions ne change pas "l'interface" fournie par L; c'est-à-dire que L continue de synthétiser $L.r$. Il faut établir quels sont les éléments dans le système de traduction qui correspondent aux éléments mentionnés dans les notes de cours: le non-terminal récursif à gauche est L, l'équivalent du non-terminal Y est la chaîne ', **num**', l'équivalent du non-terminal X est le terminal **num**, l'équivalent de l'attribut $Y.y$ est **num**.*lexval*, l'équivalent de l'attribut $X.x$ est **num**.*lexval*, la fonction g est 'postpend(rotate(·), ·)' et la fonction f est newList(·).

S	$\rightarrow$	[L]	{ $S.r := L.r$ }
L	$\rightarrow$	**num**	{ $R.i :=$ newList(**num**.*lexval*) }
		R	{ $L.r := R.s$ }
R	$\rightarrow$	, **num**	{ $R_1.i :=$ postpend(rotate($R.i$), **num**.*lexval*) }
		R_1	{ $R.s := R_1.s$ }
R	$\rightarrow$	ϵ	{ $R.s := R.i$ }

5. Dans la solution qui suit, on évite d'utiliser des attributs hérités. Le non-terminal N synthétise dans l'attribut $N.val$ la valeur numérique de la constante qu'il génère. Afin de gérer les chiffres placés après le point décimal correctement, N synthétise aussi $N.l$, qui indique le nombre de chiffres présents dans la constante. Il ne reste ensuite qu'à appliquer une formule adéquate pour calculer $R.val$. On suppose que la valeur numérique d'un chiffre a été placée au préalable dans l'attribut **digit**.*lexval* par l'analyseur lexical.

Productions	Règles sémantiques
$R \rightarrow N_1 \text{ . } N_2 \text{ \_ } N_3$	$R.val := N_1.val + \dfrac{N_2.val + \frac{N_3.val}{10^{N_3.l}-1}}{10^{N_2.l}}$
$N \rightarrow N_1$ **digit**	$N.val := 10 \times N_1.val + \textbf{digit}.lexval$ $N.l := N_1.l + 1$
$N \rightarrow$ **digit**	$N.val := \textbf{digit}.lexval$ $N.l := 1$

6. Nous attachons l'attribut t aux non-terminaux E et L. Dans le cas de $L.t$, il s'agit du type de *chacun* des éléments d'une énumération générée par L. (†) Afin de gérer le cas de l'énumération vide d'éléments, on ajoute l'attribut hérité $L.i$, lequel donne le type de l'élément immédiatement à gauche. Ainsi, une énumération vide peut prétendre assigner le même type à "chacun" de ses éléments. ($\star$) Notez qu'il faut éviter de construire un ensemble dont les éléments sont typés avec *type_error*.

Productions	Règles sémantiques
$E \rightarrow$ **cstI**	$E.t := int$
$E \rightarrow$ **cstB**	$E.t := bool$
$E \rightarrow \{\ E_1\ L\ \}$	$L.i := E_1.t \quad (\dagger)$ $E.t :=$ **if** $E_1.t = L.t \neq type\_error$ **then** $set(E_1.t)$ **else** $type\_error \quad (\star)$
$E \rightarrow \{$ **id** $\in E_1 \mid E_2 \}$	$E.t :=$ **if** $E_1.t = set(\tau)$ **and** $E_2.t = bool$ **then** $E_1.t$ **else** $type\_error$
$E \rightarrow E_1 + E_2$	$E.t :=$ **if** $E_1.t = E_2.t$ **then** $E_1.t$ **else** $type\_error$
$E \rightarrow E_1 - E_2$	$E.t :=$ **if** $E_1.t = E_2.t \neq bool$ **then** $E_1.t$ **else** $type\_error$
$E \rightarrow E_1 * E_2$	$E.t :=$ // Même calcul que pour '+'
$E \rightarrow E_1 / E_2$	$E.t :=$ **if** $E_1.t = E_2.t = int$ **then** int **else** $type\_error$
$E \rightarrow E_1 = E_2$	$E.t :=$ **if** $E_1.t = E_2.t \neq type\_error$ **then** $bool$ **else** $type\_error$
$E \rightarrow E_1 \leq E_2$	$E.t :=$ // Même calcul que pour '='
$E \rightarrow$ **pick** E_1	$E.t :=$ **if** $E_1.t = set(\tau)$ **then** τ **else** $type\_error$
$L \rightarrow$, $E\ L_1$	$L_1.i := E.t \quad (\dagger)$ $L.t :=$ **if** $E.t = L_1.t$ **then** $E.t$ **else** $type\_error$
$L \rightarrow \epsilon$	$L.t := L.i \quad (\dagger)$

7. (a) Ici, je n'ai pas cherché à produire un code particulièrement performant. Allure du code généré:

```
    E1.code
    t3 := E1.place
    E2.code
    t4 := E2.place
Lagain:
    id.place := t3
    if id.place > t4 goto Lexit
    S1.code
    t3 := t3 + 1
    goto Lagain
Lexit:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **for id** $:=$ E_1 **to** E_2 **do** S_1	$S.L_{\text{again}} :=$ **new***Label* $S.L_{\text{exit}} :=$ **new***Label* $S.t_1 :=$ **new***Temp* $S.t_2 :=$ **new***Temp* $S.code := E_1.code \parallel gen(S.t_1$ '**:=**' $E_1.place) \parallel$ $E_2.code \parallel gen(S.t_2$ '**:=**' $E_2.place) \parallel$ $gen(S.L_{\text{again}}$ ':') $\parallel gen($**id**$.place$ '**:=**' $S.t_1) \parallel$ $gen($'**if**' **id**$.place$ '>' $S.t_2$ '**goto**' $S.L_{\text{exit}}) \parallel$ $S_1.code \parallel gen(S.t_1$ '**:=**' $S.t_1$ '+' 1$) \parallel$ $gen($'**goto**' $S.L_{\text{again}}) \parallel gen(S.L_{\text{exit}}$ '**:**')

(b) Notez qu'il n'est pas spécifié *quelle* valeur vraie doit être produite lorsque l'opérateur doit produire le booléen vrai. J'ai choisi ici de préserver la valeur vraie produite par le seul des deux opérandes valant vrai mais ce n'était pas nécessaire de prendre ce soin. Je fais produire du code classique pour les deux sous-expressions. Allure du code généré:

```
    B1.code
    B2.code
    B.place := 0
    if B1.place <> 0 goto Lnext
    B.place := B2.place
Lnext:
    if B2.place <> 0 goto Lexit
    B.place := B1.place
Lexit:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \to B_1 \otimes B_2$	$B.L_{\text{next}} := \textbf{new}Label$ $B.L_{\text{exit}} := \textbf{new}Label$ $B.place := \textbf{new}Temp$ $B.code := B_1.code \parallel B_2.code \parallel gen(B.place \text{ ‘}\textbf{:=}\text{’ } 0) \parallel$ $gen(\text{‘}\textbf{if}\text{’ } B_1.place \text{‘}\neq\text{’ ‘0’ ‘}\textbf{goto}\text{’ } B.L_{\text{next}}) \parallel$ $gen(B.place \text{ ‘}\textbf{:=}\text{’ } B_2.place) \parallel gen(B.L_{\text{next}} \text{ ‘}\textbf{:}\text{’}) \parallel$ $gen(\text{‘}\textbf{if}\text{’ } B_2.place \text{‘}\neq\text{’ ‘0’ ‘}\textbf{goto}\text{’ } B.L_{\text{exit}}) \parallel$ $gen(B.place \text{ ‘}\textbf{:=}\text{’ } B_1.place) \parallel gen(B.L_{\text{exit}} \textbf{:})$

(c) Notez que je fais produire du code classique pour les deux sous-expressions. Allure du code généré:

```
    B1.code
    B2.code
    if B1.place <> 0 goto Lswap
    if B2.place = 0 goto Lfalse
    goto Ltrue
Lswap:
    if B2.place = 0 goto Ltrue
    goto Lfalse
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \to B_1 \otimes B_2$	$B.L_{\text{swap}} := \textbf{new}Label$ $B.code := B_1.code \parallel B_2.code \parallel$ $gen(\text{`}\textbf{if}\text{'}\ B_1.place \text{`}\neq\text{'}\ \text{`0'}\ \text{`}\textbf{goto}\text{'}\ B.L_{\text{swap}}) \parallel$ $gen(\text{`}\textbf{if}\text{'}\ B_2.place \text{`}=\text{'}\ \text{`0'}\ \text{`}\textbf{goto}\text{'}\ B.L_{\text{false}}) \parallel$ $gen(\text{`}\textbf{goto}'\ B.L_{\text{true}}) \parallel gen(B.L_{\text{swap}}\ \textbf{:}) \parallel$ $gen(\text{`}\textbf{if}\text{'}\ B_2.place \text{`}=\text{'}\ \text{`0'}\ \text{`}\textbf{goto}\text{'}\ B.L_{\text{true}}) \parallel$ $gen(\text{`}\textbf{goto}'\ B.L_{\text{false}})$

8. Voici l'arbre d'activation: