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

Danny Dubé
Automne 2018
Version : 22 novembre

Travail pratique #2
Traduction orientée-syntaxe → Infrastructure d'exécution

Questions

1. (*25 points.*) **Définitions orientées-syntaxe.** En vous basant sur l'une ou l'autre des grammaires suivantes, montrez comment tester l'appartenance aux langages donnés à l'aide d'une définition orientée-syntaxe. Dans chaque définition orientée-syntaxe que vous concevez, l'attribut $S.ok$ devrait être un booléen qui indique si le mot est dans le langage ou pas. Donnez une explication pour votre réponse dès que celle-ci est le moindrement complexe.

$$\begin{array}{lcl} S & \to & A \\ A & \to & \mathtt{a}\ A_1 \\ & | & \mathtt{b}\ A_1 \\ & | & \mathtt{c}\ A_1 \\ & | & \epsilon \end{array} \qquad\qquad \begin{array}{lcl} S & \to & A \\ A & \to & \mathtt{a}\ A_1 \\ & | & B \\ B & \to & \mathtt{b}\ B_1 \\ & | & C \\ C & \to & \mathtt{c}\ C_1 \\ & | & \epsilon \end{array}$$

   (a) Le langage régulier L_1 qui contient les chaînes dans lesquelles il n'y a pas de répétitions de la même lettre de longueur 2018 ou plus.

$$L_1 = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \mid w \text{ n'a pas } \mathtt{a}^{2018} \text{ ni } \mathtt{b}^{2018} \text{ ni } \mathtt{c}^{2018} \text{ comme sous-chaîne} \right\}$$

   (b) Le langage régulier L_2, lequel serait terriblement pénible à décrire à l'aide d'une expression régulière.

$$L_2 = \{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \mid w \text{ contient la sous-chaîne } \mathtt{baca} \text{ exactement 2018 fois} \}$$

   (c) Le langage régulier L_3, lequel est vraisemblablement régulier, mais qui serait terriblement difficile à décrire à l'aide d'une expression régulière. (Note : cet exercice est relativement difficile.)

$$L_3 = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \;\middle|\; \begin{array}{l} \text{il y a au moins 2018 façons différentes} \\ \text{d'extraire la sous-séquence } \mathtt{abc} \text{ de } w \end{array} \right\}$$

   (d) Le langage hors-contexte L_4. (Note : le "ou" dans la définition est inclusif.)

$$L_4 = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid 2i = j \text{ ou } 3j = k \text{ ou } 4k = i \right\}$$

   (e) Le langage L_5, lequel n'est pas hors-contexte.

$$L_5 = \left\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^k \mid i + j^2 = k^3 \right\}$$

2. (*5 points.*) **Définitions orientées-syntaxe.** Supposons qu'on a un logiciel qui lit des listes de nombres naturels à partir de l'entrée. Nos listes sont encadrées par des crochets carrés et, à l'intérieur, les nombres sont séparés par des virgules. Le logiciel doit vérifier qu'une liste qui est analysée respecte la "contrainte haut-bas" sur ses nombres. La *contrainte haut-bas* veut que les premiers nombres de la liste soient en ordre croissant (pas nécessairement strictement croissant) et que les nombres suivants soient en ordre décroissant (pas nécessairement strictement décroissant). La partie croissante de la liste peut être vide et la partie décroissante aussi.

   Vous devez concevoir une définition orientée-syntaxe à partir de la grammaire suivante. Le symbole de départ doit synthétiser l'attribut booléen S.ok qui indique si la liste analysée respecte la contrainte haut-bas ou pas.

$$
\begin{array}{lcll}
S & \rightarrow & \texttt{[ ]} & \text{// Non-terminal de départ} \\
 & | & \texttt{[} \textbf{int } L & \\
L & \rightarrow & \texttt{, } \textbf{int } L_1 & \\
 & | & \texttt{]} &
\end{array}
$$

   Par exemple, la liste `[3, 7, 12, 15, 15, 18, 17, 6, 2]` fait partie du langage. En effet, les nombres croissent jusqu'à `18`, puis ils décroissent. Au contraire, la liste `[4, 8, 11, 13, 10, 9, 21, 22, 33]` ne fait pas partie du langage puisqu'on observe une décroissance de `13` à `10` puis, plus loin à droite, une croissance de `9` à `21`.

3. (*10 points.*) **Définitions orientées-syntaxe.** Supposons que nous avons un logiciel qui manipule des *monceaux*.[1] Il s'agit d'une sorte d'arbre binaire qui sert à gérer une queue de priorité. Un monceau est un arbre binaire qui, dans cet exercice, contient des nombres naturels. Les monceaux que nous considérons ici respectent les propriétés suivantes.
   - Un monceau est *semi-ordonné*, en ce sens que le nombre contenu dans un noeud interne est supérieur aux nombres contenus dans les deux sous-arbres de ce noeud.
   - Un monceau est un arbre *complet*, en ce sens que tous les niveaux de l'arbre doivent être remplis de nombres, excepté le niveau le plus bas. Autrement dit, il y a une profondeur p telle que toutes les feuilles se situent soit à la profondeur p, soit à la profondeur $p+1$.
   - Un monceau est un arbre complet *tassé à gauche*, en ce sens que, si on fait une traversée de gauche à droite dans le monceau, on rencontre d'abord toutes les feuilles de profondeur $p+1$, puis toutes les feuilles de profondeur p. De plus, il y a au plus un noeud interne qui a un sous-arbre vide comme fils et, dans ce cas, le sous-arbre vide est le fils de droite.

   Supposons que notre logiciel a la capacité de sauvegarder des monceaux et de les relire plus tard. La syntaxe utilisée pour la *représentation externe* des monceaux est donnée par la grammaire hors-contexte ci-bas. Afin de s'assurer de l'intégrité des données lues, le logiciel doit vérifier le respect des propriétés des monceaux.

S	$\rightarrow$	T	// Non-terminal de départ
T	$\rightarrow$	`(` T_1 `/` **num** `\` T_2 `)`	// Arbre non-vide (noeud interne ou feuille)
	$\mid$	`.`	// Arbre vide

   Vous devez ajouter des attributs et des règles sémantiques à cette grammaire afin d'effectuer les vérifications demandées. Au minimum, vous devez faire synthétiser l'attribut booléen $S.ok$. L'attribut $S.ok$ indique si l'arbre respecte bien toutes les propriétés. Ajoutez tout autre attribut et toute règle sémantique qui vous apparaissent nécessaires.

   À titre d'exemple, le monceau suivant :

```
( ( (. / 2 \ .) / 8 \ (. / 7 \ .) ) / 12 \ ( (. / 3 \ .) / 9 \ .) )
```

   est valide car les trois propriétés sont respectés. Au contraire, le monceau suivant :

```
( ( (. / 6 \ .) / 3 \ (. / 1 \ .) ) / 2 \ ( (. / 4 \ .) / 9 \ .) )
```

   n'est pas valide car le nombre chez certain parents n'est supérieur aux nombres présents chez les sous-arbres. Le monceau suivant :

```
( ( ( (. / 1 \ .) / 6 \ .) / 8 \ .) / 12 \ .)
```

   n'est pas valide car il n'est pas un arbre complet. Le monceau suivant :

```
( (. / 8 \ (. / 4 \ .) ) / 12 \ (. / 9 \ .) )
```

   n'est pas valide car le noeud `8` a un sous-arbre vide à sa gauche et un sous-arbre non-vide à sa droite.

1. Une description se trouve à `https://fr.wikipedia.org/wiki/Tas_(informatique)`.

4. (*10 points.*) **Systèmes de traduction.** Considérons le système de traduction suivant. Il effectue un certain calcul sur les éléments d'une liste. Vous devez éliminer la récursion à gauche de la grammaire sous-jacente et, bien entendu, adapter les calculs faits par le système de traduction afin que le nouveau système de traduction effectue les mêmes calculs. C'est délibérément que je ne décris pas le fonctionnement de rotate, postpend et newList : ne présumez pas que ces fonctions respectent quelque propriété que ce soit (associativité, commutativité, etc.).

$$\begin{array}{lcll} S & \to & [\ L\ ] & \{\ S.r := L.r\ \} \\ L & \to & L_1\ ,\ \textbf{num} & \{\ L.r := \text{postpend}(\text{rotate}(L_1.r),\ \textbf{num}.\mathit{lexval})\ \} \\ L & \to & \textbf{num} & \{\ L.r := \text{newList}(\textbf{num}.\mathit{lexval})\ \} \end{array}$$

5. (*10 points.*) **Définitions orientées-syntaxe.** Parfois, on souhaite effectuer des calculs sur des nombres rationnels de manière exacte ; c'est-à-dire sans erreurs d'arrondissement. Une notation courante est celle des fractions ; par exemple, $\frac{4}{3}$ et $\frac{33}{14}$. Une autre notation courante est celle du développement en décimales mais en prenant soin de prévoir une façon de marquer les parties périodiques. Par exemple, strictement parlant, c'est incorrect d'écrire $\frac{4}{3} = 1.333$ car il y a une erreur introduite par l'arrondissement. Toutefois, en marquant à l'aide d'une barre qu'une partie du développement se répète pour toujours, on élimine les erreurs complètement. Par exemple, $\frac{4}{3} = 1.\overline{3}$ et $\frac{33}{14} = 2.3\overline{571428}$ sont exacts.

L'exercice de ce numéro consiste à traduire des nombres écrits en notation décimale avec partie répétitive, version source, en des nombres rationnels exactement égaux, version internalisée. Vous devez ajouter à la grammaire hors-contexte suivante des attributs et des règles sémantiques pour effectuer la conversion des nombres rationnels de la version source à la version interne en synthétisant l'attribut R.val, de type rationnel. La notation source prévoit qu'un nombre rationnel débute par une partie entière, suivie d'un point, suivi d'une partie décimale conventionnelle, suivi d'un caractère de soulignement, suivi d'une partie répétitive. Pour vous aider à calculer la valeur internalisée des nombres écrits en notation source, voici quelques identités utiles, où les a_i, b_i et c_i sont des chiffres décimaux ; i.e. de 0 à 9.

$$\begin{aligned} a_1 \ldots a_l\ .\ b_1 \ldots b_k\ \overline{c_1 \ldots c_n} &= 10 \times (a_1 \ldots a_{l-1}\ .\ a_l\ b_1 \ldots b_k\ \overline{c_1 \ldots c_n}) \\ 0\ .\ b_1 \ldots b_k\ \overline{c_1 \ldots c_n} &= \frac{b_1 + (0\ .\ b_2 \ldots b_k\ \overline{c_1 \ldots c_n})}{10} \\ 0\ .\ \overline{c_1 \ldots c_n} &= \frac{c_1 \ldots c_n}{10^n - 1} \end{aligned}$$

$$\begin{array}{rcl} R & \to & N_1\ .\ N_2\ \_\ N_3 \\ N & \to & N_1\ \textbf{digit} \\ & | & \textbf{digit} \end{array}$$

6. (*15 points.*) **Typage.** Cet exercice consiste à concevoir une définition orientée-syntaxe qui effectue le typage de diverses constructions syntaxiques d'un langage de programmation hypothétique. La syntaxe et la sémantique du langage sont décrites plus bas. Examinons le langage de programmation. Le langage permet de faire des calculs arithmétiques, booléens et ensemblistes. Dans le langage, les types suivants sont disponibles : les nombres entiers, les booléens et les ensembles. Dans le compilateur, la représentation interne du type entier est *int*, celle du type booléen est *bool* et celle du type ensemble est $set(tau)$ où τ est la représentation interne du type des éléments de l'ensemble.

   À la figure 1, on retrouve la grammaire du langage. Vous devez ajouter des attributs et des règles à la grammaire pour effectuer le typage du langage. Le non-terminal E doit synthétiser le type de l'expression dans son attribut $E.t$. Voici les règles de typage des expressions, données informellement.

   — La constante entière **cstI**. Cette expression produit un entier.
   — La constante booléenne **cstB**. Cette expression produit un booléen.
   — La construction d'un ensemble par extension $\{\ E_1\ ,\ E_2\ ,\ \ldots\ ,\ E_n\ \}$. Syntaxiquement, cette expression contient une liste de sous-expressions qui sont générées à l'aide d'une collaboration de E_1 et L. Tous les éléments de l'ensemble doivent avoir le même type. Il faut doter adéquatement les L-productions de règles afin de contribuer au typage de cette construction. (C'est d'ailleurs pour s'assurer de connaître le type de l'ensemble que notre syntaxe force la présence d'au moins un élément.)
   — La construction d'un ensemble par compréhension $\{\ \mathbf{id}\ \in\ E_1\ \mid\ E_2\ \}$. Cette expression produit un ensemble dont les éléments sont tirés de l'ensemble fourni par E_1, tout en filtrant ceux qui respectent le critère exprimé par E_2. Il faut donc que E_1 soit un ensemble, afin de pouvoir fournir des éléments. Aussi, le critère E_2 doit être du type booléen.
   — L'opération $E_1\ +\ E_2$. L'opérateur + a plusieurs usages. L'expression permet d'effectuer l'addition de deux entiers, la disjonction (le "ou") logique de deux booléens et l'union de deux ensembles. Dans chacun des trois cas, les deux opérandes doivent avoir le même type et, le cas échéant, le résultat est du même type.
   — L'opération $E_1 - E_2$. Cette expression permet de calculer la différence entre deux entiers ou entre deux ensembles. Dans chacun des deux cas, les deux opérandes doivent avoir le même type et, le cas échéant, le résultat est du même type.
   — L'opération $E_1\ *\ E_2$. Cette expression permet d'effectuer la multiplication de deux entiers et la conjonction (le "et") logique de deux booléens. Elle permet aussi de calculer l'intersection entre deux ensembles. Le typage est similaire à celui de l'opérateur +.
   — L'opération $E_1\ /\ E_2$. Cette expression permet seulement d'effectuer la division d'un entier par un autre. Le résultat est toujours un entier.
   — L'opération de comparaison $E_1 = E_2$. Cette expression permet de comparer des entiers, des booléens ou des ensembles entre eux. Dans chacun des trois cas, les deux opérandes doivent avoir le même type et, le cas échéant, le résultat est toujours du type booléen.
   — L'opération de comparaison $E_1 \leq E_2$. Cette expression permet de comparer des

E	$\rightarrow$	**cstI**	Constante entière
	$\mid$	**cstB**	Constante booléenne
	$\mid$	$\{\ E_1\ L\ \}$	Ensemble en extension
	$\mid$	$\{\ \mathbf{id}\ \in\ E_1 \mid\ E_2\ \}$	Ensemble en compréhension
	$\mid$	$E_1\ +\ E_2$	Opération binaire $+$
	$\mid$	$E_1\ -\ E_2$	Opération binaire $-$
	$\mid$	$E_1\ *\ E_2$	Opération binaire $*$
	$\mid$	$E_1\ /\ E_2$	Opération binaire $/$
	$\mid$	$E_1\ =\ E_2$	Opération de comparaison $=$
	$\mid$	$E_1\ \leq\ E_2$	Opération de comparaison $\leq$
	$\mid$	**pick** E_1	Extraction d'un élément d'un ensemble
L	$\rightarrow$	$,\ E\ L_1$	Continuation d'une ...
	$\mid$	ϵ	... énumération d'éléments

FIGURE 1 – Grammaire hors-contexte pour la syntaxe abstraite du langage.

entiers, des booléens ou des ensembles entre eux. Dans le cas d'une comparaison entre booléens, on considère que le booléen faux est plus petit que le booléen vrai. Dans le cas d'une comparaison entre ensembles, il s'agit d'un test d'inclusion. Le typage se fait de la même façon que pour l'égalité.

— L'extraction d'un élément **pick** E_1. L'opérande doit produire un ensemble. Le type du résultat sera celui des éléments du tableau.

Lorsque les règles de typage ne sont pas respectées au niveau d'une expression, il faut déclarer une erreur de types. En de telles circonstances, le type attribué à l'expression devrait être le type spécial *type_ error*, lequel existe en sus des types normaux du langage source. Ce type est produit lorsqu'une erreur de types est détectée et il doit ensuite être propagé jusqu'à la racine de l'arbre de syntaxe du programme.

Vous pouvez faire la supposition que les programmes reçus en entrée sont syntaxiquement valides ; i.e. ils respectent la grammaire. Toutefois, les programmes ne respectent pas nécessairement les règles de typage.

7. (*20 points.*) **Génération de code intermédiaire.** En vous inspirant de la génération de code intermédiaire vue en classe, vous devez créer des définitions orientées-syntaxe qui produisent le code intermédiaire pour deux construction syntaxiques. Les deux constructions sont une boucle **for** et l'opérateur logique "ou exclusif". Vous devez concevoir vos définitions orientées-syntaxe à partir des productions suivantes.

$$\begin{aligned} S &\rightarrow \textbf{for id} := E_1 \textbf{ to } E_2 \textbf{ do } S_1 \\ B &\rightarrow B_1 \otimes B_2 \end{aligned}$$

Le comportement de la boucle **for** consiste à :

(1) évaluer E_1 ; disons que la valeur est placée dans $\mathtt{t}_1$;
(2) évaluer E_2 ; disons que la valeur est placée dans $\mathtt{t}_2$;
(3) copier la valeur de $\mathtt{t}_1$ dans, disons, $\mathtt{t}_3$;
(4) copier la valeur de $\mathtt{t}_2$ dans, disons, $\mathtt{t}_4$;
(5) copier la valeur de $\mathtt{t}_3$ dans **id** ;
(6) tester **id** $> \mathtt{t}_4$ et, si le test est vrai, passer à l'étape (10) ;
(7) exécuter S ;
(8) incrémenter $\mathtt{t}_3$;
(9) passer à l'étape (5) ;
(10) point de sortie de la boucle.

Il y a quelques copies qui peuvent sembler superflues a priori. Toutefois, c'est pour éviter que le comportement de la boucle ne soit perturbé par le fait que le corps pourrait muter les variables mentionnées dans E_1 et E_2 ou la variable **id**. Par exemple, une boucle **for** $x := 2$ **to** 7 **do** S_1 doit faire passer x par les valeurs 2 à 7, successivement, même si le corps S_1 faisait entre une modification à x comme $x := 2 * x$.

L'opération logique du ou exclusif retourne vrai uniquement lorsqu'un seul des deux opérandes s'évalue à vrai.

(a) Vous devez faire produire du code intermédiaire pour la boucle **for**.

(b) Vous devez faire produire du code classique pour l'expression du ou exclusif.

(c) Vous devez faire produire du code à court-circuit pour l'expression du ou exclusif.

Attention : vous devez éviter la duplication de code ! Note : dans les sous-questions (b) et (c), vous êtes libre de faire générer du code classique ou du code à court-circuit pour les sous-expressions B_1 et B_2, selon ce qui vous facilite le plus la tâche.

8. (*5 points.*) **Infrastructure d'exécution.** Dessinez l'arbre d'activation pour l'appel $C(4, 2)$, où la fonction compte le nombre de façon de choisir 2 éléments parmi 4 ; i.e. qui calcule $\binom{4}{2}$.

$$C(n,\, k) \quad = \quad \begin{cases} 1, & \text{si } k = 0 \\ C(n-1,\, k-1) + C(n-1,\, k), & \text{si } 0 < k < n \\ 1, & \text{si } k = n \end{cases}$$

Remise des travaux

Vous devez remettre le travail via **Pixel**. Les autres modalités de remise sont inscrites dans le plan de cours.