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

Danny Dubé
Hiver 2019
Version : 11 avril

Travail pratique #2
Traduction orientée-syntaxe → Génération de code

Questions

1. (*40 points.*) **Définitions orientées-syntaxe.** En vous basant sur l'une ou l'autre des grammaires suivantes, montrez comment "reconnaître" les langages donnés à l'aide d'une définition orientée-syntaxe (DOS). Dans chaque DOS que vous concevez, l'attribut $S.ok$ devrait être un booléen qui indique si le mot est dans le langage ou pas. Aussi, illustrez comment chacune de vos DOS fonctionne bien en montrant un cas positif et un cas négatif. Dans le cas positif, choisissez un mot qui est dans le langage et montrez que votre DOS calcule une valeur vraie pour $S.ok$. Dans le cas négatif, choisissez un mot qui est hors du langage et montrez que votre DOS calcule une valeur fausse pour $S.ok$.

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

   (a) Le langage régulier L_1. (On a vu ce langage dans le TP#1 au numéro 2(a).) [1]

$$L_1 \quad = \quad \{\ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \ \mid |w|_{\mathtt{b}} \geq 1 \text{ et } |w|_{\mathtt{c}} \geq 2\}$$

   (b) Le langage régulier L_2. (On a vu un langage similaire dans le TP#1 et dans l'examen intra.)

$$L_2 \quad = \quad \{\ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \ \mid w \text{ n'a pas } \mathtt{aa}, \mathtt{bb} \text{ ou } \mathtt{cc} \text{ comme sous-chaîne}\}$$

   (c) Le langage hors-contexte L_3.

$$L_3 \quad = \quad \left\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \ \mid \ 247 + 12i = 23j + 7k\right\}$$

1. L'expression $|w|_c$ dénote le nombre d'apparitions du symbole c dans la chaîne w. C'est un peu comme l'opérateur de longueur $|\cdot|$ mais spécialisé pour le comptage d'un symbole spécifique.

(d) Le langage L_4 (qui n'est pas hors-contexte).

$$L_4 \quad = \quad \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \;\middle|\; \begin{array}{l} \text{le nombre d'apparitions de la sous-chaîne } \mathtt{bac} \\ \text{dans } w \text{ est un nombre de Fibonacci} \end{array} \right\}$$

Pour éviter des questionnements inutiles, convenons que la suite de Fibonacci est 0, 1, 1, 2, 3, 5, ...

(e) Le langage L_5 (qui n'est pas hors-contexte). Ici, la fonction $C : \Sigma_5 \to \{\mathtt{0}, \mathtt{1}\}^*$, où $\Sigma_5 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, est un code de Huffman et elle est décrite ci-bas. Par exemple, $\mathtt{ba} : \mathtt{01000}$ est dans L_5 mais pas $\mathtt{dbd} : \mathtt{101010}$, à cause du zéro superflu.

$$L_5 \quad = \quad \{f_1 \dots f_n : C(f_1) \cdot \dots \cdot C(f_n) \mid f_1 \dots f_n \in \Sigma_5^*\}$$

$$\text{où} \quad C(\mathtt{a}) = \mathtt{00}, \quad C(\mathtt{b}) = \mathtt{010}, \quad C(\mathtt{c}) = \mathtt{011}, \quad C(\mathtt{d}) = \mathtt{1}$$

Notez qu'aucune des deux grammaires suggérées ci-haut n'est adéquate pour générer les chaînes de L_5. Utilisez plutôt la grammaire suivante.

$$\begin{array}{rcl} S & \to & \mathtt{a}S_1 \mid \mathtt{b}S_1 \mid \mathtt{c}S_1 \mid \mathtt{d}S_1 \mid : T \\ T & \to & \mathtt{0}T_1 \mid \mathtt{1}T_1 \mid \epsilon \end{array}$$

2. (*15 points.*) Supposons que nous ayons un programme qui manipule des *arbres 2-3.*[2] Il s'agit d'une sorte d'arbres de recherche qui sont binaires et ternaires à la fois. Notre programme a la capacité de sauver des arbres sur disque et de les relire. Lors d'une lecture d'arbres depuis le disque, il faut s'assurer que les arbres sont syntaxiquement corrects et il faut aussi s'assurer de l'intégrité des données en vérifiant les invariants propres aux arbres 2-3. Ces invariants sont :
   — en traversant un arbre en profondeur d'abord, de gauche à droite, les *clés* (voir plus bas) que l'on retrouve dans les noeuds sont ordonnées et
   — tous les arbres vides se retrouvent exactement à la même profondeur.

   La syntaxe utilisée pour la représentation externe des arbres est donnée par la grammaire hors-contexte ci-bas. Un noeud binaire est dénoté par `[`T_1`,` x`,` T_2`]`, un noeud ternaire, par `[`T_1`,` x`,` T_2`,` y`,` T_3`]` et un arbre vide, par `[]`, où x et y sont appelées des *clés*.

   Ajoutez des règles sémantiques à la grammaire hors-contexte suivante afin que les deux invariants soient vérifiés. Le non-terminal S doit synthétiser un attribut booléen '$S.ok$' indiquant le respect ou non des invariants. Dans cet exercice, les clés ne sont que des nombres. Considérez que l'attribut **num**.*lexval* contient la valeur de la constante lue et transformée en jeton **num**.

$$
\begin{array}{lcl}
S & \rightarrow & T \\
T & \rightarrow & \texttt{[}\ T_1\ \texttt{,}\ \mathbf{num}\ \texttt{,}\ T_2\ \texttt{]} \\
 & | & \texttt{[}\ T_1\ \texttt{,}\ \mathbf{num}_1\ \texttt{,}\ T_2\ \texttt{,}\ \mathbf{num}_2\ \texttt{,}\ T_3\ \texttt{]} \\
 & | & \texttt{[]}
\end{array}
$$

   Par exemple, l'arbre suivant n'est pas valide car ses clés ne sont pas ordonnées :

```
[[[], 14, []], 12, [[], 18, []]],
```

   l'arbre suivant n'est pas valide car ses arbres vides ne sont pas tous à la même profondeur :

```
[[], 1, [[], 2, []], 3, []]
```

   mais l'arbre suivant est valide :

```
[[[], 1, []], 4, [[] 5, [], 8, []], 9, [[], 12, []]].
```

2. Pour plus de détails, voir `http://en.wikipedia.org/wiki/2-3_tree`.

3. (*15 points.*) **Systèmes de traduction.** Considérons le système de traduction suivant. Il effectue le calcul de la valeur booléenne d'un expression logique. 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. Veuillez utiliser la méthode vue en classe ; i.e. sans utiliser des raccourcis qui exploitent certaines propriétés des opérateurs logiques impliqués.

$$
\begin{array}{lcll}
E & \rightarrow & E_1 \textbf{ ou } T & \{E.b := E_1.b \vee T.b\} \\
 & | & T & \{E.b := T.b\} \\
T & \rightarrow & T_1 \textbf{ et } F & \{T.b := T_1.b \wedge F.b\} \\
 & | & F & \{T.b := F.b\} \\
F & \rightarrow & \textbf{non } F_1 & \{F.b := \neg\, F_1.b\} \\
 & | & A & \{F.b := A.b\} \\
A & \rightarrow & (\ E\ ) & \{A.b := E.b\} \\
 & | & \textbf{faux} & \{A.b := \mathsf{faux}\} \\
 & | & \textbf{vrai} & \{A.b := \mathsf{vrai}\} \\
 & | & \textbf{id} & \{A.b := \text{get\_value}(\textbf{id}.\mathit{entry})\}
\end{array}
$$

4. (*15 points.*) **Génération de code intermédiaire.** Dans le chapitre 6, nous abordons la génération du code intermédiaire pour l'énoncé de contrôle `switch`. Syntaxiquement, les énoncés `switch` comportent un certain nombre de clauses ordinaires et une clause finale obligatoire. Ici, l'exemple (a) est l'énoncé présenté au chapitre 6. On pourrait généraliser la syntaxe des énoncés `switch` en permettant d'attacher plus qu'une clé à chaque clause, comme dans l'exemple (b), et en rendant la clause finale facultative, comme dans l'exemple (c).

Exemple (a) :

switch E **with**
  case 12 : S_1 ;
  case 14 : S_2 ;
  else S_3

Exemple (b) :

switch E **with**
  case 12 : S_1 ;
  case 3, 5, 7, 11 : S_2 ;
  else S_3

Exemple (c) :

switch E **with**
  case 12 : S_1 ;
  case 3, 5, 7, 11 : S_2 ;
  case 2, 4, 6, 8 : S_3 ;
  end

En partant de la grammaire hors-contexte suivante, concevez une définition orientée-syntaxe qui produit du code intermédiaire pour les énoncés `switch` généralisés.

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{switch } E \textbf{ with } C \\
C & \rightarrow & \textbf{case } K\ S\ ;\ C_1 \\
 & | & \textbf{else } S \\
 & | & \textbf{end} \\
K & \rightarrow & \textbf{num}\ ,\ K_1 \\
 & | & \textbf{num}\ :
\end{array}
$$

5. (*5 points.*) Dessinez l'arbre d'activation pour le programme C suivant. La racine de l'arbre doit être l'activation de la fonction principale.

```
#include <stdio.h>

int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-2) + fib(n-1);
}

void main(void)
{
  printf("fib(4) = %d\n", fib(4));
  return;
}
```

6. (*10 points.*) Établissez les durées de vie des différentes variables dans le code intermédiaire suivant et effectuez l'allocation des registres. Tâchez de placer les variables dans le plus petit nombre de registres possible. Considérez que les variables `a`, `b`, `c`, `z1` et `z2` sont des variables qui proviennent directement du programme source et qu'elles ont par conséquent une durée de vie illimitée et qu'elles doivent être allouées en mémoire.

Code interm.

1. t_1 := uminus b
2. t_2 := b * b
3. t_3 := 4 * a
4. t_4 := t_3 * c
5. t_5 := t_2 - t_4
6. t_6 := sqrt t_5
7. t_7 := 2 * a
8. t_8 := t_1 + t_6
9. z1 := t_8 / t_7
10. t_9 := t_1 - t_6
11. z2 := t_9 / t_7

Remise des travaux

Vous devez remettre le travail via **monPortail** d'ici le **23 avril**. Veuillez ignorer la date de remise indiquée dans le plan de cours. Les autres modalités de remise sont inscrites dans le plan de cours.