Introduction à la compilation – Examen (1ère session)

Durée : 3 heures

Tout document autorisé.

Université Paris Diderot – Licence 3

(2011-2012)

Le soin apporté à la rédaction et à la présentation ainsi que la rigueur des réponses seront pris en compte dans la notation. Ce sujet se décompose en 2 parties indépendantes. La première est une application directe du cours d'analyse syntaxique. La suivante porte sur des extensions du langage de programmation étudié en travaux dirigés.

A. Analyse syntaxique

Soit l'ensemble de terminaux $T = \{\lambda, ., (,), x\}$. On définit la grammaire G du non terminal M par les règles de production suivantes :

$$\begin{array}{rcll} M & \to & x & (1) \\ & | & M\,M & (2) \\ & | & \lambda x.M & (3) \\ & | & (M) & (4) \end{array}$$

Exercice 1 (Ambiguïté)

1. *Quels sont les deux arbres de production de la grammaire G associé au mot $\lambda x.xx$?*
2. *Supposons que l'on se donne la règle de résolution d'ambiguïté suivante :*

   « La règle (2) a une priorité plus forte que la règle (3). »

   Lequel des deux arbres de production de la question précédente serait alors accepté ?
3. *Calculez "$FIRST(M)$" et "$FOLLOW(M)$".*
4. *Calculez la table LL(1) de cette grammaire. Comment se traduit l'ambiguité de la grammaire en termes de conflit(s) LL(1) ?*

□

On se donne une nouvelle grammaire G' pour représenter le langage du non-terminal M :

$$\begin{array}{rcll} M & \to & \lambda x.M & (M_1) \\ & | & N & (M_2) \end{array}$$

$$N \to x \qquad (N_1)$$

3. *Pourquoi la grammaire G' n'est-elle pas LL(1) ?*
4. *Transformez la grammaire G' en une grammaire équivalente et LL(1). Donnez la table LL(1) correspondante. Comment avez-vous résolu la dernière ambiguïté relevée dans la question 2 ?*
5. *On souhaite une associativité à gauche pour la règle N_2. Est-ce que votre analyseur syntaxique l'autorise ? Si non, comment transformer votre analyseur syntaxique pour qu'il produise un arbre de syntaxe abstraite respectant cette règle d'associativité à gauche ?*

□

B. Théorie des langages de programmation

Nous considérons un langage de programmation impératif simple dont la syntaxe abstraite est spécifiée par la grammaire suivante :

e	::=	**Expression**
	n	Entier
	x	Variable
	$e \oplus e$	Opération binaire
c	::=	**Commande**
	$x := e$	Affectation
	skip	Commande vide
	$c; c$	Séquencement
	while e **do** c	Boucle non bornée
	if e **then** c **else** c	Branchement

L'opérateur $\oplus$ dénote une opération binaire parmi $+$, $-$, $/$, $*$ et $>$.

Exercice 3 (Syntaxe)

1. *Après avoir rappelé le type OCaml permettant de représenter les arbres de syntaxe abstraite de la grammaire précédente, donnez une valeur OCaml de ce type représentant le programme P suivant :*

$$a := 42;$$
$$b := 5;$$

Ce langage est impératif : la sémantique d'un programme est la transformation d'un état initial, formé d'une mémoire, en un nouvel état, dans lequel les valeurs associées aux variables peuvent avoir été modifiées et de nouvelles variables ont pu être allouées. On modélise une mémoire de la même façon qu'un environnement, à l'aide d'un dictionnaire M dont la syntaxe est :

$$
\begin{array}{rcll}
M & ::= & & \textbf{Mémoire} \\
 & | & \bullet & \text{Mémoire vide} \\
 & | & M + \{x \mapsto n\} & \text{Mémoire où la variable } x \text{ vaut } n
\end{array}
$$

On rappelle les règles du jugement « $M(x) = n$ » qui se lit « Dans la mémoire M, la variable x a la valeur n. » :

$$\frac{}{(M + \{x \mapsto n\})(x) = n} \qquad \frac{M(x) = n \qquad x \neq y}{(M + \{y \mapsto n'\})(x) = n}$$

Exercice 4 (Opérations sur la mémoire)

1. *Parmi les jugements suivants, lesquels sont dérivables ? Vous justifierez votre réponse en donnant l'arbre de dérivation des jugements dérivables et une explication informelle pour les jugements non dérivables :*
   (a) $((\bullet + \{x \mapsto 42\}) + \{y \mapsto 0\})(x) = 42$.
   (b) $((\bullet + \{x \mapsto 42\}) + \{y \mapsto 0\})(y) = 42$
   (c) $((\bullet + \{x \mapsto 42\}) + \{y \mapsto 0\})(z) = 42$
   (d) $((\bullet + \{x \mapsto 42\}) + \{x \mapsto 0\})(x) = 42$
2. *Dans la sémantique que nous avons donnée en cours, l'affectation d'une valeur n à une variable x dans une mémoire M s'effectue en construisant la mémoire $M + \{x \mapsto n\}$. Même si cette spécification est adéquate, elle est particulièrement inefficace car la taille de la mémoire est proportionnelle aux nombres d'affectations effectuées durant l'évaluation du programme. On se propose donc de définir un jugement plus économe en mémoire pour représenter des affectations. On l'écrit "$M[x \leftarrow n] \Downarrow M'$. Il se lit "$M'$ est la mémoire M dans laquelle la valeur de x a été modifiée en n." Les règles qui définissent ce jugement sont :*

$$\frac{}{\bullet[x \leftarrow n] \Downarrow \bullet + \{x \mapsto n\}} \qquad \frac{}{(M + \{x \mapsto m\})[x \leftarrow n] \Downarrow (M + \{x \mapsto n\})}$$

$$\frac{M[x \leftarrow n] \Downarrow M' \qquad x \neq y}{(M + \{y \mapsto m\})[x \leftarrow n] \Downarrow (M' + \{y \mapsto m\})}$$

Est-ce que le jugement

$$(\bullet + \{x \mapsto 0\})[x \leftarrow 42] \Downarrow (\bullet + \{x \mapsto 42\})$$

est dérivable ? Si oui, donnez l'arbre de dérivation correspondant.

3. *Donnez le code OCaml d'une fonction qui attend une mémoire M, un identificateur x et un entier n, puis calcule M' telle que $M[x \leftarrow n] \Downarrow M'$.*

□

On rappelle aussi les règles de sémantique à grands pas du jugement « $M \vdash c \Downarrow M'$ » qui se lit « La commande c transforme la mémoire M en la mémoire M'. » :

$$\frac{}{M \vdash \textbf{skip} \Downarrow M} \qquad \frac{M \vdash e \Downarrow n}{M \vdash x := e \Downarrow M + \{x \mapsto n\}} \qquad \frac{M \vdash c_1 \Downarrow M_1 \qquad M_1 \vdash c_2 \Downarrow M_2}{M \vdash c_1; c_2 \Downarrow M_2} \qquad \frac{M \vdash e \Downarrow 0}{M \vdash \textbf{while } e \textbf{ do } c \Downarrow M}$$

$$\frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c;\ \textbf{while } e \textbf{ do } c \Downarrow M'}{M \vdash \textbf{while } e \textbf{ do } c \Downarrow M'} \qquad \frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c_1 \Downarrow M'}{M \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \Downarrow M'}$$

$$\frac{M \vdash e \Downarrow 0 \qquad M \vdash c_2 \Downarrow M'}{M \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \Downarrow M'}$$

Exercice 5 (Sémantique à grands pas)

1. *Modifiez la règle d'évaluation des affectations pour qu'elle s'appuie sur le jugement $M[x \leftarrow n] \Downarrow M'$ défini dans l'exercice précédent.*
2. *Donnez la ou les règles d'évaluation pour la construction* **do** c **while** e.
3. *Donnez la ou les règles d'évaluation pour la construction* **for** $x := e_1$ **to** e_2 **do** c.
4. *Étendez la fonction OCaml d'évaluation des commandes vue en cours pour prendre en compte les deux règles précédentes.*
5. *Pour pouvoir étendre la sémantique à grands pas aux constructions* **break** *et* **continue**, *on étend la forme des résultats du calcul. Désormais le jugement de sémantique à grands pas des commandes est de la forme $M \vdash c \Downarrow R$ avec :*

R ::=		***Résultat***
	M	*Évaluation normale en une mémoire M*
|	**Stop** (M)	*Boucle interrompue*
|	**Continue** (M)	*Itération interrompue*

Les deux résultats Stop et Continue font changer de "mode" d'interprétation. En effet, dès qu'une sous-commande s'évalue ainsi tous les calculs suivants sont ignorés jusqu'à la prochaine boucle englobante. Les règles d'inférence suivantes mettent en pratique cette idée :

$$\textsc{Continue}\ \frac{}{M \vdash \textbf{continue} \Downarrow \textbf{Continue}\ (M)} \qquad \textsc{Break}\ \frac{}{M \vdash \textbf{break} \Downarrow \textbf{Stop}\ (M)}$$

$$\textsc{Full-Seq}\ \frac{M \vdash c_1 \Downarrow M_1 \qquad M_1 \vdash c_2 \Downarrow M_2}{M \vdash c_1; c_2 \Downarrow M_2}$$

$$\textsc{Stop-Seq}\ \frac{M \vdash c_1 \Downarrow R \qquad R = \textbf{Stop}\ (M') \text{ ou } \textbf{Continue}\ (M')}{M \vdash c_1; c_2 \Downarrow R}$$

$$\textsc{Full-While}\ \frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c \Downarrow M' \qquad M' \vdash \textbf{while}\ e\ \textbf{do}\ c \Downarrow M''}{M \vdash \textbf{while}\ e\ \textbf{do}\ c \Downarrow M''}$$

$$\textsc{Stop-While}\ \frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c \Downarrow \textbf{Stop}\ (M')}{M \vdash \textbf{while}\ e\ \textbf{do}\ c \Downarrow M'}$$

$$\textsc{Continue-While}\ \frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c \Downarrow \textbf{Continue}\ (M') \qquad M' \vdash \textbf{while}\ e\ \textbf{do}\ c \Downarrow M''}{M \vdash \textbf{while}\ e\ \textbf{do}\ c \Downarrow M''}$$

$$\textsc{If-True}\ \frac{M \vdash e \Downarrow n \qquad n \neq 0 \qquad M \vdash c_1 \Downarrow R}{M \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \Downarrow R} \qquad \textsc{If-False}\ \frac{M \vdash e \Downarrow 0 \qquad M \vdash c_2 \Downarrow R}{M \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \Downarrow R}$$

Expliquez chacune de ces règles.

6. *Donnez les nouvelles règles de la construction* **do** c **while** e.
7. *Adaptez la fonction OCaml d'évaluation des commandes pour traiter le* **continue** *et le* **break**.

□