Examen Analyse Syntaxique et Compilation

Université Paris 7. Licence d'Informatique (L3). Deuxième session 2007-08. Durée 2h.

L'utilisation de documents ou de dispositifs électroniques est interdite.

25 juin 2008

Exercice 1 (15 points) *Cet exercice comporte 7 questions. On se place dans le cadre (d'un fragment) du langage impératif étudié dans le cours auquel on ajoute les commandes skip, fail et catch(S, S'). La syntaxe des commandes est spécifiée par la grammaire G ci-dessous :*

$$S \to id := n \mid S; S \mid skip \mid fail \mid catch(S, S)$$

(1) *Montrez que la grammaire G est ambiguë.*

(2) *Proposez une grammaire équivalente non-ambiguë.*

Considérez une grammaire simplifiée G' avec productions $S \to a \mid S; S$. Soit L' le langage généré par la grammaire G' et soit $L'' = \{w\$ \mid w \in L'\}$ le langage constitué des mots dans L' auxquels on ajoute à la fin un symbole '\$'.

(3) *Expliquez pourquoi la grammaire G'' suivante qui génère le langage L'' n'est pas LR(0) :*

$$S \to S'\$ \qquad S' \to a \mid a; S'$$

(4) *Proposez une grammaire équivalente à G'' qui est LR(0) (en expliquant pourquoi).*

L'évaluation des commandes (spécifiées par la grammaire G) est décrite par un jugement de la forme $(S, \eta, \mu) \Downarrow (X, \mu')$ où μ, μ' sont des mémoires et $X \in \{fail, skip\}$ indique si le calcul termine normalement (skip) ou si une exception (non-capturée) a été levée (fail). Les règles d'évaluation sont les suivantes :

$$\frac{X \in \{fail, skip\}}{(X, \eta, \mu) \Downarrow (X, \mu)} \qquad \frac{}{(x := n, \eta, \mu) \Downarrow (skip, \mu'[n/\eta(x)])}$$

$$\frac{(S_1, \eta, \mu) \Downarrow (skip, \mu') \quad (S_2, \eta, \mu') \Downarrow (X, \mu'')}{(S_1; S_2, \eta, \mu) \Downarrow (X, \mu'')} \qquad \frac{(S_1, \eta, \mu) \Downarrow (fail, \mu')}{(S_1; S_2, \eta, \mu) \Downarrow (fail, \mu')}$$

$$\frac{(S_1, \eta, \mu) \Downarrow (skip, \mu')}{(catch(S_1, S_2), \eta, \mu) \Downarrow (skip, \mu')} \qquad \frac{(S_1, \eta, \mu) \Downarrow (fail, \mu') \quad (S_2, \eta, \mu') \Downarrow (X, \mu'')}{(catch(S_1, S_2), \eta, \mu) \Downarrow (X, \mu'')}$$

(5) *Appliquez les règles d'évaluation aux commandes ci-dessous à partir d'un environnement η_0 qui associe aux variables a, b, c, d les locations distinctes $\ell_1, \ell_2, \ell_3, \ell_4$ et une mémoire μ_0 qui*

associe aux locations ℓ_i, $i = 1, 2, 3, 4$, *la valeur* 5.

$$S_1 = catch(a := 1; fail; b := 2\ ,\ c := 3)$$
$$S_2 = catch(a := 1; fail\ ,\ catch(b := 2; fail; c := 3\ ,\ d := 4))$$

Dans le cours nous avons discuté une compilation du langage impératif vers un code octet dont on rappelle de suite certaines instructions.

- build n 0 *: on écrit l'entier* n *au sommet de la pile et on incrémente le compteur ordinal.*
- load n *: on copie l'*n*-ième valeur de la pile au sommet de la pile (de valeurs) et on incrémente le compteur ordinal.*
- goto j *: on affecte* j *au compteur ordinal.*
- read *si la valeur au sommet de la pile est une location* ℓ*, on remplace* ℓ *par son contenu et on incrémente le compteur ordinal.*
- write *si les valeurs au sommet de la pile sont* $v \cdot \ell$ *alors on écrit* v *dans la location* ℓ *et on élimine* v *et* ℓ*.*
- stop *: on arrête le calcul avec succès.*
- fail *: on arrête le calcul dans un état d'échec.*

(6) *Définissez une fonction de compilation* $\mathcal{C}(S, w, \kappa, \kappa')$ *où* S *est la commande à compiler,* w *est une liste de variables,* κ *est l'adresse à laquelle continuer le calcul si l'évaluation de* S *termine normalement et* κ' *est l'adresse à laquelle continuer le calcul si l'évaluation de* S *lève une exception.*

(7) *Calculez la compilation des commandes* S_1 *et* S_2 *décrites ci-dessus avec paramètres,* $w = abcd$, $\kappa = 100$, $\kappa' = 200$ *et en supposant que la première instruction est mémorisée à l'adresse* 1.

Exercice 2 (5 points) *Considérez la session OCAML suivante et pour chaque ligne déterminez le type et éventuellement la valeur numérique calculée par OCAML.*

```
let f1 = function x -> function y -> x ;;
let f2 = function x -> function y -> y ;;

let f3 = function f -> function x -> f(f x) ;;
let f4 = function f -> function x -> f( f (f x) ) ;;

let f5 = function b -> function x -> function y -> (b x) y ;;
let f6 = function x -> x + 34 ;;

let f7 = f3 f6 ;;
let f8 = f4 f6 ;;

let f9 = f5 f1 f7 f8 54 ;;
let f10 = f5 f2 f7 f8 54 ;;
```

Corrigé

Exercice 1

1. Le mot *skip*; *skip*; *skip* admet 2 arbres de dérivation.
2. On peut éliminer l'ambiguïté en décidant, par exemple, que la séquentialisation associe à gauche :
$$S \to E \mid E; S$$
$$E \to id := n \mid skip \mid fail \mid catch(S, S)$$
3. En construisant l'AFN des items on a :
$$q_0 \xrightarrow{\epsilon} S \to \cdot S'\$ \xrightarrow{\epsilon} S' \to \cdot a \xrightarrow{a} S' \to a\cdot$$
$$q_0 \xrightarrow{\epsilon} S \to \cdot S'\$ \xrightarrow{\epsilon} S' \to \cdot a; S' \xrightarrow{a} S' \to a\cdot; S'$$
Donc dans la version déterminisée on aura un état qui contient un item complet ($S' \to a\cdot$) et un autre item ($S' \to a\cdot; S'$).
Une autre façon de se rendre compte du problème est d'analyser, par exemple, une dérivation LR du mot $a; a$. Quand on a a sur la pile on a un conflit *shift-reduce*. Faut-il faire le *shift* du prochain symbole (;) ou le *reduce* de a ($S' \to a$) ?
4. Une grammaire LR(0) est
$$S \to S'\$$$
$$S' \to S'; a \mid a$$
On construit l'automate des items, on le déterminise et on vérifie qu'un état qui contient un item complet ne contient pas d'autres items. Il est instructif de vérifier que le conflit shift-reduce a disparu. Si on a juste a sur la pile on applique un *reduce* ($S' \to a$) et si on a $S'; a$ alors on applique un *reduce* ($S' \to S'; a$).
5. L'évaluation de S_1 exécute les affectations $a := 1$ et $c := 3$. Donc à la fin de l'évaluation, les locations ℓ_1 et ℓ_3 contiennent les valeurs 1 et 3 respectivement. L'évaluation de S_2 exécute les affectations $a := 1$, $b := 2$ et $d := 4$. Donc à la fin de l'évaluation, les locations ℓ_1, ℓ_2 et ℓ_4 contiennent les valeurs 1, 2 et 4, respectivement.
6. Une fonction de compilation possible est :
$$\begin{aligned}
\mathcal{C}(id := n, w, \kappa, \kappa') &= (\mathsf{build}\ n)(\mathsf{load}\ i(id, w))(\mathsf{write})(\mathsf{goto}\ \kappa) \\
\mathcal{C}(S; S', w, \kappa, \kappa') &= \mathcal{C}(S, w, \kappa_1, \kappa')\kappa_1 : \mathcal{C}(S', w, \kappa, \kappa') \\
\mathcal{C}(skip, w, \kappa, \kappa') &= (\mathsf{goto}\ \kappa) \\
\mathcal{C}(fail, w, \kappa, \kappa') &= (\mathsf{goto}\ \kappa') \\
\mathcal{C}(catch(S, S'), w, \kappa, \kappa') &= \mathcal{C}(S, w, \kappa, \kappa_1)\kappa_1 : \mathcal{C}(S', w, \kappa, \kappa')
\end{aligned}$$
7. En remplaçant les adresses symboliques par des adresses numériques, la compilation de S_1 est :
```
1:  build 1
2:  load  1
3:  write
4:  goto 5
5:  goto 10
6:  build 2
7:  load 2
```

```
8:  write
9:  goto 100
10: build 3
11: load 3
12: write
13: goto 100
```

et celle de S_2 est :

```
1:  build 1
2:  load  1
3:  write
4:  goto 5
5:  goto 6
6:  build 2
7:  load 2
8:  write
9:  goto 10
10: goto 15
11: build 3
12: load 3
13: write
14: goto 100
15: build 4
16: load 4
17: write
18: goto 100
```

Exercice 2

Voici la session OCAML avec des commentaires sur la signification des fonctions :

```
let f1 = function x -> function y -> x ;;            (* première prj *)
let f2 = function x -> function y -> y ;;            (* deuxième prj *)
let f3 = function f -> function x -> f(f x) ;;       (* itère 2 fois *)
let f4 = function f -> function x -> f( f (f x) ) ;; (* itère 3 fois *)
let f5 = function b ->
         function x -> function y -> (b x) y ;;      (* une sorte d'ite *)
let f6 = function x -> x + 34 ;;                     (* une fonct. num. *)
let f7 = f3 f6 ;;                                    (* pour itérer 2 fois f6 *)
let f8 = f4 f6 ;;                                    (* idem 3 fois *)
let f9 = f5 f1 f7 f8 54 ;;                           (* selectionne f7 et lui passe 54 *)
let f10 = f5 f2 f7 f8 54 ;;                          (* idem pour f8 *)

# val f1 : 'a -> 'b -> 'a = <fun>
# val f2 : 'a -> 'b -> 'b = <fun>
# val f3 : ('a -> 'a) -> 'a -> 'a = <fun>
# val f4 : ('a -> 'a) -> 'a -> 'a = <fun>
# val f5 : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# val f6 : int -> int = <fun>
# val f7 : int -> int = <fun>
# val f8 : int -> int = <fun>
# val f9 : int = 122
# val f10 : int = 156
```