Examen Analyse Syntaxique et Compilation

Université Paris 7. Licence d'Informatique (L3). Première session 2007-2008. Durée 2h.

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

8 mai 2008

Exercice 1 (6 points) *On considère la grammaire G suivante avec S symbole initial et ϵ mot vide :*

$$\begin{array}{lcl} S & \rightarrow & (A) \\ A & \rightarrow & \epsilon \mid id \mid id, B \\ B & \rightarrow & id \mid id, B \end{array}$$

1. *Montrez que G n'est pas LL(1).*
2. *Montrez que G n'est pas LR(0).*
3. *Trouvez une grammaire LL(1) équivalente à G (c'est-à-dire qui génère le même langage).*
4. *Expliquez pourquoi la grammaire que vous proposez est LL(1).*
5. *Trouvez une grammaire LR(0) équivalente à G.*
6. *Expliquez pourquoi la grammaire que vous proposez est LR(0).*

Rappel *Pour montrer qu'une grammaire est LL(1) on vérifie certaines conditions sur null, First et Follow et pour montrer qu'une grammaire est LR(0) on vérifie certaines conditions sur l'automate des items déterminisé.*

Exercice 2 (5 points) *On se place dans le cadre (d'un fragment) du langage impératif étudié dans le cours. On ajoute les commandes skip, fail et $catch(S, S')$:*

$$S ::= id := e \mid S; S \mid skip \mid fail \mid catch(S, S')$$

Informellement, la commande skip ne fait rien, la commande fail 'lève une exception' et la commande $catch(S, S')$ entraîne l'exécution de la commande S ; si l'exécution de cette commande lève une exception alors on procède à l'exécution de la commande S'. Par exemple, les commandes suivantes entraînent toutes l'exécution des affectations $x := 3$, $y := 4$ et $z := 5$ mais pas de $w := 6$:

$$\begin{array}{l} x := 3; y := 4; z := 5; fail; w := 6 \\ catch(x := 3; fail; w := 6 \ , \ y := 4; z := 5) \\ catch(x := 3; fail \ , \ catch(y := 4; fail; w := 6 \ , \ z := 5)) \end{array}$$

Pour évaluer les expressions, on suppose un jugement $(e, \eta, \mu) \Downarrow (n, \mu')$, où n est un nombre entier, η est un environnement et μ et μ' sont des mémoires.

Pour formaliser le comportement de fail et catch on introduit un jugement de la forme :

$$(S, \eta, \mu) \Downarrow (X, \mu')$$

où $X \in \{fail, skip\}$ *indique si le calcul termine normalement (skip) ou si une exception (non-capturée) a été levée (fail). On pose comme axiome :*

$$\frac{X \in \{fail, skip\}}{(X, \eta, \mu) \Downarrow (X, \mu)}$$

Proposez des règles d'évaluation pour : (1) l'affectation, (2) la séquentialisation et (3) le catch. Vérifiez que les règles proposées capturent le comportement souhaité des commandes fail et catch dans les exemples ci-dessus.

Suggestion *Il s'agit ici de généraliser les règles étudiées dans le cours pour l'affectation et la séquentialisation et de concevoir des règles pour le catch.*

Exercice 3 (5 points) (1) *Considérez la session OCAML suivante et pour chaque ligne déterminez le type et éventuellement la valeur calculée par OCAML.*

```
let f = function h -> function x -> h (h x) ;;
let g = f f ;;
let s = function x -> x+1 ;;
let d = function x -> x+x ;;
let c = function x -> x*x ;;
g s 2 ;;
g d 2 ;;
g c 2 ;;
```

(2) *On se place dans le cadre du langage fonctionnel dont on rappelle les règles de typage :*

$$\frac{E(x) = \tau}{E \vdash x : \tau} \quad \frac{E[\tau/x] \vdash e : \tau'}{E \vdash \lambda x.e : \tau \to \tau'} \quad \frac{E \vdash e : \tau \to \tau' \quad E \vdash e' : \tau}{E \vdash ee' : \tau'}$$

Trouvez une expression sans variables libres avec le type ci-dessous et présentez la preuve de typage associée.

$$(\tau_1 \to (\tau_2 \to \tau_3)) \to ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3))$$

Exercice 4 (4 points) *On imagine que les blocs de mémoire libres sont organisés dans une* liste libre *pointée par* `LL`. *Les adresses des blocs dans la liste libre sont* croissants. *Le premier mot de chaque bloc pointé par* `p` *comporte un champ avec la longueur du bloc (indiqué par* `p.long`*) et un champ qui pointe au bloc suivant de mémoire libre (indiqué par* `p.next`*). On utilise un pointeur* `nil` *pour représenter une liste vide.*

On souhaite implémenter une commande `free(p)` *qui a l'effet de 'libérer' un bloc de mémoire pointé par* `p` *en l'insérant dans la liste pointée par* `LL`. *Au passage, on veut compacter le bloc libéré avec des blocs de mémoire libre contigus (s'ils existent).*

Par exemple, en supposant que la liste libre ait la forme à gauche, l'effet de `free(15)` *avec* `15.long=3` *doit être de générer une liste libre avec la forme à droite :*

	long	*next*
10 :	(4,	18)
18 :	(5,	25)
25 :	(7,	*nil*)

	long	*next*
10 :	(4,	15)
15 :	(8,	25)
25 :	(7,	*nil*)

Décrivez un algorithme pour implémenter la commande free.

Corrigé

Exercice 1

(1) La grammaire G n'est pas LL(1). On considère par exemple les productions $A \to id$ et $A \to id, B$. Clairement, $\mathit{First}(id) \cap \mathit{First}(id, B) = \{id\} \neq \emptyset$.

(2) La grammaire G n'est pas LR(0). En effet, dans la construction de l'automate des items on a les transitions :

$$\begin{array}{lcl} A \to id, \cdot B & \xrightarrow{\epsilon} & B \to \cdot id \\ A \to id, \cdot B & \xrightarrow{\epsilon} & B \to \cdot id, B \\ B \to \cdot id & \xrightarrow{id} & B \to id\cdot \\ B \to \cdot id, B & \xrightarrow{id} & B \to id\cdot, B \end{array}$$

Ainsi, dans l'AFD on a un état qui contient au moins les items $B \to id\cdot$ (item complet !) et $B \to id\cdot, B$.

Remarque Le langage généré par G est *rationnel* (ou régulier). Par exemple, on peut spécifier le langage par la grammaire linéaire droite G' suivante :

$$\begin{array}{l} S \to (A \\ A \to) \mid idB \\ B \to) \mid , idB \end{array}$$

(3) G' est LL(1).

(4) En effet il n'y a pas d'élément nullable et si $X \to \alpha \mid \beta$ sont deux productions différentes alors les premiers symboles de α et β sont deux terminaux différents.

(5) G' est LR(0).

(6) On construit l'automate des items et on le déterminise pour vérifier qu'un état qui contient un item complet ne contient pas d'autre item. Voici les transitions de l'AFN :

$$\begin{array}{l} q_0 \xrightarrow{\epsilon} S \to \cdot(A \\ S \to \cdot(A \xrightarrow{(} S \to (\cdot A \xrightarrow{A} S \to (A\cdot \\ S \to (\cdot A \xrightarrow{\epsilon} A \to \cdot) \quad S \to (\cdot A \xrightarrow{\epsilon} A \to \cdot idB \\ A \to \cdot) \xrightarrow{)} A \to)\cdot \\ A \to \cdot idB \xrightarrow{id} A \to id \cdot B \xrightarrow{B} A \to idB\cdot \\ A \to id \cdot B \xrightarrow{\epsilon} B \to \cdot) \\ A \to id \cdot B \xrightarrow{\epsilon} B \to \cdot, idB \\ B \to \cdot, idB \xrightarrow{,} B \to , \cdot idB \xrightarrow{id} B \to , id \cdot B \xrightarrow{B} B \to , idB\cdot \\ B \to , id \cdot B \xrightarrow{\epsilon} B \to \cdot) \\ B \to , id \cdot B \xrightarrow{\epsilon} B \to \cdot, idB \end{array}$$

Exercice 2

1. L'exécution d'une affectation entraîne une terminaison normale (*skip*) :

$$\frac{(e, \eta, \mu) \Downarrow (v, \mu')}{(x := e, \eta, \mu) \Downarrow (skip, \mu'[v/\eta(x)])}$$

2. Pour la séquentialisation il faut distinguer 2 situations selon que la première commande s'évalue en *skip* ou en *fail* :

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

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

3. Pour le *catch* aussi on distinguera 2 situations :

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

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

Exercice 3

1. Voici la session OCAML :

```
# let f = function h -> function x -> h (h x) ;;
val f : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = f f ;;
val g : ('_a -> '_a) -> '_a -> '_a = <fun>
# let s = function x -> x+1 ;;
val s : int -> int = <fun>
# let d = function x -> x+x ;;
val a : int -> int = <fun>
# let c = function x -> x*x ;;
val m : int -> int = <fun>
# g s 2 ;;
- : int = 6
# g d 2 ;;
- : int = 32
# g c 2 ;;
- : int = 65536
```

2. On peut prendre $\lambda x.\lambda y.\lambda z.xz(yz)$, soit en OCAML :

```
# let f = function x -> (function y -> (function z -> x z ( y z) ) ) ;;
val f : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Exercice 4

On définit une fonction `compact(p,l)` qui insère le bloc pointé par `p` dans la liste pointée par `l` en respectant l'ordre et en compactant les blocs. La fonction retourne le pointeur à la liste compactée.

```
compact(p,l) = case
l=nil            : p.next:=nil; p
p<l,p+p.long<l   : p.next:=l  ; p
p<l,p+p.long=l   : p.next:=l.next; p.long:=p.long+l.long; p
p>l              : let l1 = compact(p,l.next) in
                   if l+l.long = l1 then
                                   l.long:= l.long+l1.long;
                                   l.next:= l1.next
                   l
```

La liste libre peut être vide. Par ailleurs, on fait l'hypothèse que les blocs dans la liste libre et le bloc libéré peuvent être contigus mais qu'ils ne se chevauchent pas. Si `LL` est le pointeur à la liste libre et `p` est le pointeur au bloc libéré, on obtient l'effet souhaité en exécutant :

```
LL:= compact(p,LL)
```