Université Montpellier II
Maîtrise d'Informatique
Module Langage, Evaluation, Compilation — 3 heures

R. Ducournau – M. Lafourcade

9 septembre 2003

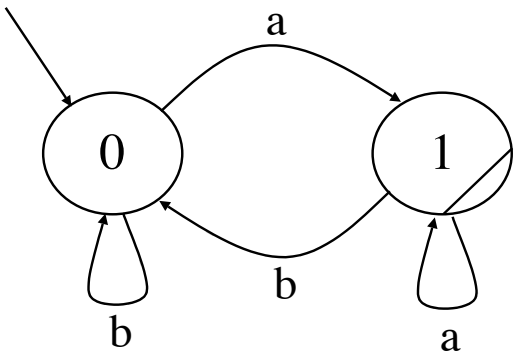
Documents autorisés : notes de cours, polys, pas de livre.
2 parties indépendantes, notées chacune sur 10.

1 Compilation d'automates

Nous disposons d'une machine virtuelle (VM) analogue à celle du cours. L'objectif est de générer, dans le langage de cette VM, le code correspondant à l'interprétation d'automates. On suppose que la VM est adaptée aux listes LISP, avec des opérations `CAR` et `CDR` pour récupérer le contenu d'une cellule, ainsi que `CONSP` et `NULL` pour vérifier que le contenu d'un registre est une cellule ou la liste vide (NIL).

Question 1

Ecrire le code de la VM correspondant à l'automate de la figure :



La donnée (mot à reconnaître) est une liste de caractères (symboles) dans le registre R0. A l'issue du l'exécution, R0 contient l'état final ou NIL suivant que le mot a été reconnu ou pas, et l'état de la pile doit être inchangé.

Indiquez comment vous traduisez les états de l'automate dans la VM. Avez-vous besoin de la pile ? Pourquoi ?

On dispose, en LISP, d'un type de données abstrait automate, muni de l'interface fonctionnelle suivante :

(auto-etats auto) retourne la liste des états (entiers) de l'automate : pour l'automate de la figure, (0 1)

(auto-init auto) retourne l'état (entier) initial de l'automate ;

(auto-final-p auto etat) retourne vrai si l'état argument est final ;

(auto-trans auto etat) retourne la liste des transitions issues de l'état argument, sous la forme d'une liste d'association dont les clés sont les caractères et les valeurs associées la liste des états que l'on peut atteindre par une transition d'origine l'état argument et de caractère la clé.

Pour l'état 0 et l'automate de la figure, on obtiendrait ((a 1) (b 0)).

Question 2

L'objectif est d'écrire la fonction LISP `auto2vm` qui retourne le code VM correspondant à un automate (c'est-à-dire un code voisin de celui que vous avez écrit pour la question précédente et l'automate de la figure).

1. Spécifier le principe de la génération : comment traduire les états, les transitions, les états finaux, l'état initial, etc.
2. Décomposer le problème en définissant des fonctions annexes pour traiter séparément, chaque transition, chaque état, etc.
3. Faire d'abord une version pour les automates déterministes (rappeler la définition). La pile est-elle nécessaire ?
4. Indiquer les différences dans le cas des automates non déterministes. La pile est-elle nécessaire ?
5. que faut-il faire pour que la fonction `auto2vm` retourne le code VM d'une fonction à un argument, le mot à reconnaître, et retournant l'état final ou NIL ?

Question 3

On veut maintenant générer une fonction LISP avec une fonction `auto2lisp`. Sans écrire complètement `auto2lisp`, spécifier le principe de la génération : comment traduire les états, les transitions, les états finaux, l'état initial, etc. compte-tenu du langage cible, LISP.

2 Méta-évaluateur paresseux

Le méta-évaluateur vu en cours est un évaluateur habituel, déterministe et « glouton » (*eager* en anglais) qui évalue tout, même s'il n'en a pas besoin effectivement. L'objectif de cette question est de réaliser, ou au moins de spécifier, un méta-évaluateur « paresseux » (*lazy*) qui n'évalue une expression que si c'est nécessaire.

La différence entre ces 2 modes d'évaluation est plus précisément la suivante :

– un évaluateur glouton évalue tous les arguments d'une fonction avant de l'appliquer ;
– un évaluateur paresseux n'évalue aucun argument d'une fonction interprétée avant d'en avoir besoin.

Exemple Soit la fonction

```
(defun foo (w x y z)
       (if (< w x) y z))
```

La durée de l'évaluation de l'expression `(foo 1 2 (fact n) (fibo n))` sera en $O(n)$ avec un évaluateur paresseux et en $O(2^n)$ avec un évaluateur glouton (en supposant `fibo` écrite de façon naïvement récursive).

Question 4

Différence de comportement :

1. dans quelles conditions le comportement de ces 2 types d'évaluateurs sera-t-il certainement identique, ou au contraire risque-t-il d'être différent ?
2. expliquer ce comportement identique à l'aide du λ-calcul.

La différence entre les deux modes d'évaluation se ramène d'abord à une différence dans la structure des environnements. Dans un évaluateur glouton, l'environnement lexical est constitué d'une liste d'association variable-valeur, par exemple `((w . 1)(x . 2)(y . 24)(z . 5))` pour l'exemple précédent, si l'on suppose que l'appel de `foo` s'est fait dans l'environnement `((n . 4))`.

Avec l'évaluateur paresseux, l'environnement lexical doit être une structure plus compliquée où

– chaque variable est liée, au début, à l'expression argument non évaluée, et à l'environnement dans lequel il faudrait l'évaluer ;
– dès que l'évaluation de la variable a été forcée, la variable doit être liée à la valeur résultant de cette évaluation ;

Question 5

Spécification des environnements :

1. proposer une structure de données qui permette de représenter cet environnement à géométrie variable ; ne pas oublier qu'il faut être capable de faire la différence entre l'expression évaluée et non évaluée ;
2. spécifier et définir la fonction LISP `make-lazy-env` qui construit un environnement paresseux à partir des arguments d'un appel et des paramètres de la fonction (ne pas traiter `&optional`, etc.) ;
3. spécifier et définir la fonction `lazy-meval-symbol` qui évalue une variable dans un tel environnement paresseux.
4. spécifier et définir la fonction `lazy-setf-symbol` qui affecte une variable dans un tel environnement paresseux (discutez de la pertinence d'autoriser une affectation dans un tel évaluatateur).

Question 6

Spécifier les modifications qu'il est nécessaire d'apporter au corps de la fonction `meval`, pour la transformer en `lazy-meval`, ainsi que dans les fonctions annexes, pour rendre paresseux le méta-évaluateur du cours. Préciser a contrario les parties qui ne changent pas, lorsque c'est significatif.