Université Montpellier II
Master d'Informatique - Spécialité AIGLE
GMIN 108 Compilation Interprétation — 2 heures

R. Ducournau – M. Lafourcade

janvier 2012

Documents autorisés : notes de cours, polys, pas de livre.

1 Compilation d'automates vers une VM (sur 15)

Nous disposons d'une machine virtuelle (VM) proche de celle du cours mais très simplifiée. L'objectif est de générer, un code de cette VM permettant l'interprétation d'automates déterministes, c'est-à-dire la reconnaissance d'un mot par un automate.

On suppose que la VM peut gérer des listes LISP, avec des opérations spécialisées :

- `(car R1 R2)` prend la cellule dont l'adresse est dans le registre `R1` et charge son champ `car` dans le registre `R2` ; (`cdr` a le rôle symétrique pour le champ `cdr`) ;
- l'opération de comparaison `(cmp R1)` permet d'effectuer des tests de cellules (`consp`, `atom`, `null` en LISP) sur le contenu du registre `R1` en positionnant des flags de manière usuelle et
- `(bconsp #label)` est une instruction de branchement conditionnel qui effectue le branchement si le drapeau préalablement positionné indique qu'il s'agit bien d'une cellule ; `batom` est le branchement conditionnel complémentaire et `bnull` le branchement conditionné au fait que la valeur testée est `nil`.

Les conventions pour le code d'interprétation des automates sont les suivantes. La donnée (mot à reconnaître) est une liste de caractères (symboles) contenue dans le registre `R0`. A l'issue de l'exécution, la VM s'arrête et `R0` contient l'état final atteint lors de l'exécution de l'automate ou `nil` suivant que le mot a été reconnu ou pas.

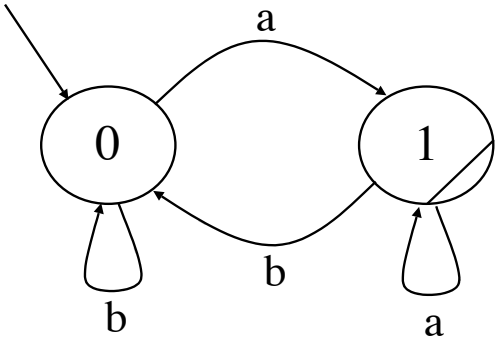
Question 1

Pour éviter les ambiguités, commencez par définir précisément et succinctement les instructions de la VM dont vous vous servez.

(NB traitez cette question à la fin, pour ne pas avoir à décrire toutes les instructions de la VM.)

Question 2

Soit l'automate déterministe suivant, dont l'état initial est 0 et l'unique état final est 1 :



1. Commencez par indiquer comment il est possible de traduire les états de l'automate dans la VM. Une pile est-elle nécessaire ? Pourquoi ?
2. Ecrire le code de la VM correspondant à l'automate donné en exemple ci-dessus, en le commentant.

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

(auto-etat-liste 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-tran-list auto etat) retourne la liste des transitions issues de l'état argument, sous la forme d'une liste d'associations 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 3

Écrire la fonction LISP `auto2vm` qui retourne le code VM correspondant à un automate déterministe (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.

Question 4

Indiquer dans les grandes lignes comment il faudrait généraliser ça à des automates non-déterministes ?

Rappelons qu'un automate est déterministe si, de chaque état, chaque caractère de l'alphabet considéré conduit au plus à une transition.

2 Analyse statique du code intermédiaire (sur 8)

La transformation de LISP vers le langage intermédiaire (LI) est une occasion de faire différentes vérifications syntaxiques et sémantiques. Certaines vérifications ne sont possibles qu'en LISP (sur les variables par exemples) mais d'autres sont toujours possibles dans le code du LI.

Question 5

Vérification du nombre des arguments dans les appels de fonctions méta-définies (`::mcall`).

1. Ecrire la fonction de parcours d'arbre du LI qui vérifie que chaque appel de fonction méta-définie a le bon nombre d'arguments ;
2. Que faut-il faire dans le cas de `:unknown` ?
   NB à la réflexion, il n'y a qu'une seule chose à faire !

Question 6

Liste des fonctions appelées : étant donnée une fonction méta-définie `foo`, on souhaite calculer la liste des fonctions méta-définies qui sont appelées, directement ou indirectement, par `foo`.

1. Ecrire la fonction `compute-callee` qui prend en argument un symbole (le nom d'une fonction méta-définie) et retourne la liste des fonctions méta-définies appelées indirectement par la fonction argument.

   La fonction `compute-callee` appelle une fonction auxiliaire `compute-callee-LI` qui implémente le parcours d'arbre.

   On simplifiera l'écriture de ces fonctions en les traitant en récursion semi-terminale, en ajoutant un paramètre supplémentaire qui désigne la liste qu'on est en train de construire.

   Attention aux boucles infinies !
2. Que faut-il faire dans le cas de `:unknown` ?