Université Montpellier II — Master d'Informatique
FMIN 106 Compilation — 2 heures

R. Ducournau – M. Lafourcade

Janvier 2011

Documents autorisés : notes de cours, polys, pas de livre.
Notation globale sur 23.

Le point de départ de ce sujet est le langage intermédiaire du méta-évaluateur "threadé" (Chapitre 5 du polycopié "Compilation et Interprétation des Langages"). On suppose la transformation effectuée (par la fonction appelée `mtrans` dans le poly et `lisp2li` dans le cours) et on s'intéresse ici à des transformations, écrites en Lisp, d'expressions du langage intermédiaire.

On se restreindra à la partie du langage dont la syntaxe des *expressions évaluables* (`<expr-eval-li>`) est la suivante :

<expr-eval-li>	:=	« (**:const .** » <expr> «) » |
		« (**:var .** » <int> «) » |
		« (**:if** » <expr-eval-li> <expr-eval-li> « **.** » <expr-eval-li> «) » |
		« (**:progn** » <expr-eval-li> <expr-eval-li>+ «) » |
		« (**:set-var** » <int> « **.** » <expr-eval-li> «) » |
		« (**:mcall** » <symbol> <expr-eval-li>* «) » |
		« (**:call** » <symbol> <expr-eval-li>* «) » |
		« (**:unknown** » <expr-eval-lisp> « **.** » <env> «) »

Il y a donc, dans l'ordre, des mots-clés pour : les constantes et les variables, une forme conditionnelle, la séquence, l'affectation de variable, l'appel de fonction méta-définie, et de fonction prédéfinie, et enfin le cas des expressions encore inconnues au moment de la transformation.

Par rapport à Lisp, le langage est grandement simplifié : il n'y a plus de `let` (la transformation est censée les avoir fait disparaître, suivant le principe décrit dans le poly) et on ne s'intéresse pas ici à des constructions de plus haut niveau comme des fermetures, les fonctions locales ou les mots-clés `&rest` ou `&optional`. Il n'y a plus de macros—sauf dans la forme `:unknown` quand elle concerne une macro définie après la transformation du code considéré.

La définition d'une fonction méta-définie, c'est-à-dire sa "valeur fonctionnelle", est constituée d'une paire contenant l'expression unique qui forme le corps de la fonction et un entier qui indique la taille de l'environnement à créer (ou la réservation de pile à faire) lors de l'appel.

1 Exemple de langage intermédiaire (sur 6)

Question 1

Ecrire la définition LISP de la fonction `length` qui calcule la longueur d'une liste, sous deux formes : en récursion terminale (`len-term`) et enveloppée (`len-env`).

Attention : pour la version terminale, ne pas utiliser de fonction récursive locale avec `labels`, mais uniquement une fonction globale, car la transformation dans le langage intermédiaire serait trop difficile.

En donner ensuite la valeur fonctionnelle dans le langage intermédiaire, sous les deux formes terminales et enveloppées.

Question 2

Donner pour `len-term` et `len-env`, le nombre d'expressions évaluables présentes dans leur définition du langage intermédiaire.

Question 3

Ecrire la fonction LISP `max-depth-li` qui retourne la profondeur maximum d'une expression évaluable dans la syntaxe du langage intermédiaire. Ecrire l'analyse par cas explicite correspondant à la grammaire de la page précédente.

2 Appels terminaux (sur 10)

Rappelons qu'un appel de fonction est terminal s'il n'est pas enveloppé, c'est-à-dire si la fonction appelante retourne le résultat renvoyé par la fonction appelée sans effectuer le moindre traitement entre le retour de l'appelée et le retour de l'appelant. C'est une généralisation de la notion de récursion terminale. Tout appel terminal n'est pas récursif. Inversement, une fonction récursive enveloppée contient certainement un appel terminal.

On généralise avec la notion de sous-expression terminale. Soit une expression e du langage intermédiaire. Une sous-expression f de e est terminale dans e si l'évaluation de e se termine par l'évaluation de f en retournant sa valeur, sans qu'aucun traitement ne soit effectué entre le retour de e et celui de f.

Question 4

1. Montrer que cette notion est transitive : si g est terminale dans f et que f est terminale dans e, alors g est terminale dans e.

   En déduire qu'un appel (`:call` ou `:mcall`) est terminal si c'est une sous-expression terminale de toutes les expressions qui le contiennent.

2. Dans chacun des cas suivants d'expressions évaluables, indiquer pour chaque sous-expression si elle est terminale ou pas.

   « **(:if** » <expr-eval-li-1> <expr-eval-li-2> « **.** » <expr-eval-li-3> « **)** » |
   « **(:progn** » <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n> « **)** » |
   « **(:set-var** » <int> « **.** » <expr-eval-li> « **)** » |
   « **(:mcall** » <symbol> <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n>« **)** » |
   « **(:call** » <symbol> <expr-eval-li-1> <expr-eval-li-2> ... <expr-eval-li-n> « **)** » |
   « **(:unknown** » <expr-lisp> « **.** » <env> « **)** »

   Est-il possible qu'il y ait 2 réponses positives pour le même cas ? Expliquer pourquoi.

On étend la grammaire du langage avec le mot-clé `:mcallt` pour les appels terminaux de fonctions méta-définies et `:callt` pour les appels terminaux de fonctions prédéfinies.

Question 5

Modifier les définitions des fonctions `len-term` et `len-env` (question 1) dans le langage intermédiaire de façon à utiliser le mot-clé approprié pour chaque appel.

Question 6

Spécifier et écrire la fonction LISP `marque-terminal-li` qui transforme la valeur fonctionnelle d'une fonction en langage intermédiaire en se servant de `:callt` et `:mcallt` pour les appels terminaux.

Traiter spécifiquement les différents cas de récursion de `marque-terminal-li`, comme dans la Question 3.

3 Génération de code (sur 7)

Question 7

Expliquer comment peut être réalisé un appel terminal dans le code assembleur de la machine virtuelle.

Question 8

Ecrire le code assembleur de la fonction factorielle terminale, en tenant compte du fait qu'il s'agit d'un appel terminal.