École Normale Supérieure

Langages de programmation et compilation

examen 2015–2016

Jean-Christophe Filliâtre

27 janvier 2016

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
Les deux problèmes sont indépendants. L'épreuve dure 3 heures.

---

1 Allocation de registres pour des arbres

Dans ce problème, on s'intéresse au nombre de registres nécessaires pour évaluer une expression arithmétique réduite à des variables, des négations et des additions. De telles expressions sont décrites par la syntaxe abstraite suivante :

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & -e & \text{négation} \\ & | & e+e & \text{addition} \end{array}$$

La machine vers laquelle on compile ces expressions contient des registres et une adresse pour chaque variable. Les instructions de cette machine sont les suivantes :

`load`	x, r	lit à l'adresse x	$(r \leftarrow mem[x])$
`neg`	r	négation	$(r \leftarrow -r)$
`add`	r_1, r_2	addition	$(r_2 \leftarrow r_1 + r_2)$

où x désigne une adresse et r, r_1, r_2 des registres. Étant donnée une expression e et un registre r, on cherche à produire une séquence d'instructions qui place la valeur de e dans le registre r. Les règles du jeu de ce problème sont qu'on ne s'autorise pas à effectuer des simplifications sur l'expression (comme $x + y + -x = y$), ni à exploiter l'associativité de l'addition. En revanche, on peut exploiter la commutativité pour évaluer e_2 avant e_1, plutôt que e_1 avant e_2, quand on évalue $e_1 + e_2$.

Question 1 Donner, pour chacune des expressions suivantes, le nombre minimal de registres nécessaires à son calcul (en incluant le registre cible) :

1. $((x + y) + z) + t$
2. $(x + y) + (z + t)$
3. $x + ((y + z) + t)$

Question 2 On note $n(e)$ le nombre minimal de registres nécessaires au calcul de l'expression e (en incluant le registre cible). Définir $n(e)$ par récurrence sur l'expression e.

Question 3 On définit la taille $t(e)$ d'une expression e comme le nombre d'additions qu'elle contient, c'est-à-dire $t(x) = 0$, $t(-e) = t(e)$ et $t(e_1+e_2) = 1+t(e_1)+t(e_2)$. Minorer $t(e)$ par une expression impliquant $n(e)$. (On demande une preuve de cette inégalité.) En déduire la taille minimale d'une expression dont ne peut pas calculer la valeur avec seulement 5 registres.

Compilation. On suppose que la machine contient un nombre fini $K \geq 2$ de registres, notés $r_1, \ldots, r_K$. Pour compiler des expressions arithmétiques de taille arbitraire, on suppose que la machine fournit une pile, non bornée, que l'on peut manipuler avec deux nouvelles instructions :

`push`	r	empile la valeur du registre r
`pop`	r	dépile une valeur, dans le registre r

Question 4 Définir une fonction *compile*(e) qui renvoie une suite d'instructions plaçant la valeur de e dans le registre r_1. La pile ne doit pas être utilisée lorsque $n(e) \leq K$.

---

2 Les Inoxydables Symboles Parenthésés

Dans ce problème, on étudie LISP, un langage étonnant de simplicité. Un programme LISP est réduit à une expression. Une expression ne prend que trois formes possibles : un symbole (toute suite de caractères autres que les parenthèses et ne commençant pas par un chiffre), une constante entière (toute suite de chiffres) ou une liste possiblement vide d'expressions, écrite entre parenthèses.

e	::=	x	symbole
	|	n	constante entière
	|	`(` e ... e `)`	liste

Pour donner un sens aux programmes LISP, certains symboles seront interprétés de façon particulière lors de l'évaluation ; nous l'expliquerons plus loin.

Grammaire. Pour effectuer l'analyse syntaxique du langage LISP, on se donne la grammaire suivante :

E	$\to$	`sym`
	|	`int`
	|	`(` L `)`
L	$\to$	ϵ
	|	$E\ L$

L'ensemble des terminaux est $\{$`sym`, `int`, `(`, `)`$\}$ et l'ensemble des non terminaux est $\{E, L\}$. Le symbole de départ est E et il correspond à une expression LISP.

Question 5 Cette grammaire est-elle LL(1) ? LR(0) ? SLR(1) ? LR(1) ? Justifier à chaque fois.

Analyse syntaxique. On se donne le type OCaml suivant pour la syntaxe abstraite des expressions LISP :

```
type expr = Sym of string | Num of int | Lst of expr list
```

On suppose par ailleurs avoir écrit un analyseur lexical, qui construit des lexèmes dans le type OCaml suivant :

```
type token = SYM of string | NUM of int | LPAR | RPAR
```

où `SYM` dénote un symbole, `NUM` une constante entière, `LPAR` une parenthèse ouvrante et `RPAR` une parenthèse fermante.

Question 6 Écrire une fonction OCaml `expr: token list -> expr * token list` qui reconnaît exactement une expression LISP au début de la liste passée en argument et renvoie cette expression ainsi que les lexèmes non consommés. Si une telle expression n'existe pas, cette fonction doit lever l'exception `SyntaxError`.

Sémantique. Si une liste e contient au moins un élément, on note $car(e)$ son premier élément et $cdr(e)$ la liste de ses autres éléments. (Si e n'est pas une liste, ou une liste vide, la signification de $car(e)$ et de $cdr(e)$ n'est pas spécifiée.) Si e_2 est une liste, on note $cons(e_1, e_2)$ la liste dont le premier élément est e_1 et dont les autres éléments sont les éléments de e_2. (Si e_2 n'est pas une liste, la signification de $cons(e_1, e_2)$ n'est pas spécifiée.)

Les listes sont mutables, au sens où un élément d'une liste peut être remplacé par une autre expression. En particulier, si la liste e contient au moins un élément, on note $car(e) \leftarrow e_1$ le remplacement du premier élément de e par e_1.

Une *liste d'association* est une liste de listes de longueur 2, de la forme `((`x_1 e_1`) (`x_2 e_2`) ... (`x_n e_n`))`, où les x_i sont des symboles. Si a est une liste d'association, on note $assoc(x, a)$ le premier élément de a de la forme `(`x e`)`, le cas échéant, et `()` sinon. Un *environnement* est une liste non vide `(`a_1 a_2 ... a_m`)` de listes d'associations. Si k est un environnement, on note $lookup(x, k)$ le premier résultat de $assoc(x, a_i)$ qui n'est pas `()`, le cas échéant, et `()` sinon.

La valeur d'une expression LISP est une expression LISP. L'évaluation d'une expression e se fait dans un environnement k, qui donne la valeur des variables. On note $[e]_k$ le résultat de cette évaluation. Sa définition est donnée figure 1. Elle donne une interprétation particulière à certains symboles, à savoir cinq symboles qui demandent une évaluation spécifique

`quote` `if` `define` `begin` `lambda`

et plusieurs symboles correspondant à des opérations primitives, à savoir

`+` `-` `*` `/` `=` `<` `<=` `>` `>=` `cons` `car` `cdr`

Noter que la définition de $[e]_k$ est partielle : certains comportements ne sont pas spécifiés. L'évaluation d'un programme e consiste à évaluer l'expression e dans un environnement qui est initialisé à `(())`, c'est-à-dire une liste contenant une unique liste d'association, qui est vide.

Question 7 Donner la valeur de chacune des expressions suivantes :

1. `(begin (define x 42) (define x 43) x)`
2. `((lambda (x) (quote x)) 42)`
3. `(begin (define x 12) ((lambda (y) (define x y)) 42) x)`

Question 8 Donner un exemple d'expression pour laquelle la sémantique n'est pas spécifiée.

Question 9 Représenter l'environnement après l'évaluation de l'expression suivante :

```
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
```

Expliquer par quel mécanisme la fonction `fib` ainsi définie peut effectivement être récursive. Si on évalue ensuite l'expression `(fib 4)`, illustrer l'environnement juste après l'appel.

Question 10 Est-il possible de définir des fonctions mutuellement récursives ? Justifier.

expression e	évaluation $[e]_k$
n (constante entière)	renvoyer n
x (symbole)	renvoyer $car(cdr(lookup(x,\ k)))$
`(quote` e_1`)`	renvoyer e_1

`(if` e_1 e_2 e_3`)`	si $[e_1]_k$ est différent de `()` alors
	  renvoyer $[e_2]_k$
	sinon
	  renvoyer $[e_3]_k$

`(define` x e_1`)`	faire $car(k) \leftarrow cons(($ `(`x $[e_1]_k$`)`$,\ car(k))$ renvoyer `()`
`(begin` e_1 ... e_n`)`	évaluer $[e_1]_k$, ..., $[e_{n-1}]_k$ renvoyer $[e_n]_k$
`(`p e_1 ... e_n`)`	renvoyer le résultat de la primitive p sur les valeurs $[e_1]_k$, ..., $[e_n]_k$
`(lambda` e_1 e_2`)`	renvoyer `(`e_1 e_2 k`)`
`(`e_1 e_2 ... e_n`)`	si $[e_1]_k$ est une liste `(`p b k_b`)` et si p est une liste de symboles `(`x_2 ... x_n`)` soit a la liste `((`x_2 $[e_2]_k$`)` ... `(`x_n $[e_n]_k$`))` renvoyer $[b]_{cons(a,\ k_b)}$

primitive p	arité	sémantique
`+`, `-`, `*`, `/`	2	attend deux entiers et renvoie un entier (le résultat du calcul)
`=`, `<`, `<=`, `>`, `>=`	2	attend deux entiers et renvoie le symbole `t` si la comparaison est positive et `()` sinon
`cons`	2	*cons*
`car`, `cdr`	1	*car*, *cdr*

FIGURE 1 – Sémantique de LISP.

Question 11 Donner la définition en LISP d'une fonction `map` et d'une fonction `range` de telle sorte que le programme suivant

```
(begin
  (define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
  (define map ...)
  (define range ...)
  (map fib (range 0 10))
)
```

calcule la liste des 10 premières valeurs de la suite de Fibonacci. La fonction `map` doit prendre deux arguments f et l et renvoyer une liste obtenue en appliquant la fonction f à tous les éléments de l. La fonction `range` doit prendre en arguments deux entiers a et b et renvoyer la liste de tous les entiers compris entre a inclus et b exclus.

Interprète LISP en OCaml. On s'intéresse maintenant à l'écriture d'un interprète LISP en OCaml. Pour tenir compte du caractère mutable des listes, on modifie le type OCaml donné plus haut pour la syntaxe abstraite de LISP, de la manière suivante :

```
type expr =
| Sym  of string
| Num  of int
| Nil
| Cons of expr ref * expr
```

Le constructeur `Nil` représente la liste vide et une liste $(e_1 \dots e_2)$ est maintenant représentée par l'expression OCaml `Cons (ref` e_1, `Cons (ref` e_2, ..., `Cons (ref` e_n, `Nil)` ... `))`.

Question 12 Écrire des fonctions OCaml `cons`, `car` et `cdr` sur ce type.

Question 13 On souhaite écrire une fonction `eval: expr -> expr -> expr` qui prend en arguments un environnement k et une expression e et renvoie l'expression $[e]_k$ lorsqu'elle existe. Donner l'extrait du code de cette fonction correspondant aux cas de la construction `define`, de la construction `lambda` et de l'application (dernière ligne du tableau 1). On ne cherchera pas à traiter les cas d'erreurs (le programmeur LISP ne fait pas d'erreur).

Interprète LISP en assembleur. On s'intéresse maintenant à l'écriture d'un interprète LISP en assembleur X86-64.

Question 14 Proposer une représentation en mémoire des expressions LISP pour un tel interprète.

Question 15 Donner le code assembleur des fonctions `cons`, `car` et `cdr`, avec les conventions d'appel usuelles de X86-64.

Question 16 Donner le code assembleur correspondant à l'évaluation de la construction `if`. On supposera que l'expression à évaluer se trouve dans le registre `%rdi` et que l'environnement se trouve dans le registre `%rsi`.

Interprète LISP en LISP. L'une des particularités de LISP est qu'il n'y a pas de distinction entre la syntaxe abstraite et les valeurs. On peut donc espérer pouvoir écrire un interprète LISP en LISP.

Question 17 Discuter la possibilité d'un tel interprète, c'est-à-dire la possibilité de définir une fonction LISP de la forme

```
(define eval (lambda (e k) ...))
```

telle que `(eval` e k`)` évalue l'expression e dans l'environnement k. En particulier, on indiquera s'il est nécessaire d'ajouter de nouvelles primitives pour y parvenir.

Question 18 Donner l'invocation de la fonction `eval` pour évaluer l'expression `(+ 40 2)`.

Annexe : aide-mémoire X86-64

On donne ici un fragment du jeu d'instructions X86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur X86-64. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`mov` r_2`,` r_1	copie le registre r_2 dans le registre r_1
`mov` n`,` r_1	charge la constante n dans le registre r_1
`mov` L`,` r_1	charge la valeur à l'adresse L dans le registre r_1
`mov` $\$L$`,` r_1	charge l'adresse de l'étiquette L dans le registre r_1
`add` r_2`,` r_1	calcule la somme de r_1 et r_2 dans r_1 (on a de même `sub` et `imul`)
`mov` n`(`r_2`),` r_1	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`push` r_1	empile la valeur contenue dans r_1
`pop` r_1	dépile une valeur dans le registre r_1
`cmp` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 - r_2$
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 \& r_2$
`je` L	saute à l'adresse désignée par l'étiquette L en cas d'égalité (on a de même `jne`, `jg`, `jge`, `jl` et `jle`)
`jmp` L	saute à l'adresse désignée par l'étiquette L
`call` L	saute à l'adresse désignée par l'étiquette L, après avoir empilé l'adresse de retour
`ret`	dépile une adresse et y effectue un saut

On alloue de la mémoire sur le tas avec un appel à `malloc`, qui attend un nombre d'octets dans `%rdi` et renvoie l'adresse du bloc alloué dans `%rax`.