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. $x + ((y + z) + t)$

Correction : L'idée est d'effectuer les calculs en commençant toujours par la sous-expression nécessitant le plus de registres.

1. 2, par exemple $(((x1 + y2)1 + z2)1 + t2)1$
2. 3, par exemple $((x1 + y2)1 + (z2 + t3)2)1$
3. 2, par exemple $(x2 + ((y1 + z2)1 + t2)1)1$

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.

Correction : On traduit l'idée consistant à commencer systématiquement par la sous-expression nécessitant le plus de registres.

$$\begin{array}{rcll} n(x) & = & 1 & \\ n(-e) & = & n(e) & \\ n(e_1+e_2) & = & 1+n(e_1) & \text{si } n(e_1)=n(e_2) \\ & = & \max(n(e_1),n(e_2)) & \text{sinon} \end{array}$$

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.

Correction : On montre par récurrence sur e que

$$t(e) \geq 2^{n(e)-1} - 1.$$

En effet, si $e = x$ alors $t(e) = 0 \geq 2^{1-1} - 1 = 0$. Si $e = -e_1$, alors $t(e) = t(e_1)$ et $n(e) = n(e_1)$ et l'hypothèse de récurrence permet de conclure. Enfin, si $e = e_1 + e_2$, on distingue deux cas :

– si $n(e_1) = n(e_2)$ alors

$$\begin{array}{rcll} t(e) & = & 1 + t(e_1) + t(e_2) & \\ & \geq & 1 + 2^{n(e_1)-1} - 1 + 2^{n(e_2)-1} - 1 & \text{par H.R.} \\ & = & 2^{n(e)-1} - 1. & \end{array}$$

– si $n(e_1) \neq n(e_2)$, supposons sans perte de généralité que $n(e_1) > n(e_2)$. Alors

$$\begin{array}{rcll} t(e) & = & 1 + t(e_1) + t(e_2) & \\ & \geq & 1 + 2^{n(e_1)-1} - 1 + 2^{n(e_2)-1} - 1 & \text{par H.R.} \\ & > & 2^{n(e_1)-1} - 1 & \text{car } 2^{n(e_2)-1} > 0 \\ & = & 2^{n(e)-1} - 1. & \end{array}$$

On en déduit que s'il faut au moins 6 registres, c'est-à-dire $n \geq 6$, la taille est au moins égale à 31.

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 $n(e) \leq K$.

Correction : On commence par écrire une fonction plus générale, prenant en arguments une expression e et un registre k, plaçant la valeur de e dans r_k. L'idée est que les registres inférieurs à k ne sont plus disponibles. On maintient l'invariant $1 \leq k \wedge k + \min(2, n(e)) \leq K + 1$.

$$
\begin{array}{rcl}
compile(x, k) & = & \texttt{load}\ x, r_k \\
compile(-e, k) & = & compile(e, k);\ \texttt{neg}\ r_k \\
compile(e_1 + e_2, k) & = & \text{si } n(e_1) \geq n(e_2) \text{ alors} \\
 & & \quad \text{si } k + 1 + \min(2, n(e_2)) \leq K + 1 \text{ alors} \\
 & & \quad\quad compile(e_1, k);\ compile(e_2, k+1);\ \texttt{add}\ r_{k+1}, r_k \\
 & & \quad \text{sinon} \\
 & & \quad\quad compile(e_1, k);\ \texttt{push}\ r_k;\ compile(e_2, k);\ \texttt{pop}\ r_{k+1};\ \texttt{add}\ r_{k+1}, r_k \\
 & & \text{sinon} \\
 & & \quad compile(e_2 + e_1, k)
\end{array}
$$

La fonction demandée est alors $compile(e, 1)$.

---

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.

$$
\begin{array}{rcll}
e & ::= & x & \text{symbole} \\
 & | & n & \text{constante entière} \\
 & | & \texttt{(}\ e\ \ldots\ e\ \texttt{)} & \text{liste}
\end{array}
$$

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 :

$$
\begin{array}{rcl}
E & \to & \texttt{sym} \\
 & | & \texttt{int} \\
 & | & \texttt{(}\ L\ \texttt{)} \\
L & \to & \epsilon \\
 & | & E\ L
\end{array}
$$

L'ensemble des terminaux est $\{\texttt{sym}, \texttt{int}, \texttt{(}, \texttt{)}\}$ 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.

Correction : On a clairement $\text{NULL}(E) = \texttt{false}$ et $\text{NULL}(L) = \texttt{true}$. On calcule facilement $\text{FIRST}(E) = \text{FIRST}(L) = \{\texttt{sym}, \texttt{int}, \texttt{(}\}$, $\text{FOLLOW}(E) = \{\texttt{sym}, \texttt{int}, \texttt{(}, \texttt{)}, \#\}$. et $\text{FOLLOW}(L) = \{\texttt{)}\}$.

La table d'expansion est

	sym	int	(	)	#
E	sym	int	(L)		
L	EL	EL	EL	ϵ	

donc la grammaire est LL(1).

Dans l'automate LR(0), on a notamment l'état

$$\begin{array}{lcl} E & \to & \texttt{(} \bullet L\texttt{)} \\ L & \to & \bullet \\ L & \to & \bullet EL \\ E & \to & \bullet\texttt{sym} \\ E & \to & \bullet\texttt{int} \\ E & \to & \bullet\texttt{(}L\texttt{)} \end{array}$$

qui contient un conflit lecture/réduction (lecture de `sym` et réduction de $L \to \epsilon$). Donc la grammaire n'est pas LR(0). Ce conflit disparaît dans l'analyse SLR(1), car $\text{FOLLOW}(L) = \{\texttt{)}\}$. De même, on a un conflit lecture/réduction dans l'état LR(0)

$$\begin{array}{lcl} L & \to & E \bullet L \\ L & \to & \bullet \\ L & \to & \bullet EL \\ E & \to & \bullet\texttt{sym} \\ E & \to & \bullet\texttt{int} \\ E & \to & \bullet\texttt{(}L\texttt{)} \end{array}$$

qui disparaît dans l'analyse SLR(1) pour la même raison. Les autres états de l'automate LR(0), au nombre de 7, ne contiennent pas de conflit (ni lecture/réduction, ni réduction/réduction). La grammaire est donc SLR(1).

La grammaire est LR(1) par inclusion car elle est LL(1) (ou SLR(1), au choix).

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`.

Correction : On l'écrit de façon mutuellement récursive avec une seconde fonction, `tail`, qui reconnaît une liste d'expressions jusqu'à trouver une parenthèse fermante (qui est consommée).

```
let rec expr = function
| SYM s :: input -> Sym s, input
| NUM n :: input -> Num n, input
| LPAR  :: input -> tail [] input
| [] | RPAR :: _ -> raise SyntaxError

and tail acc = function
| RPAR :: input -> Lst (List.rev acc), input
| input         -> let e, input = expr input in tail (e :: acc) input
```

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.

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 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)`

Correction :

1. `43`
2. `x`
3. `12`

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

Correction : `(1 2)`

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.

Correction : L'environnement k est de la forme `(`a`)` où a est la liste d'association

`((fib ((n) (if ...)` k`)))`.

Il contient donc une référence à k lui-même, ce qui permet à la fonction d'être récursive. En effet, un appel à `(fib 4)` va construire un nouvel environnement de la forme

`(((n 4))` a`)`

Le corps de `fib` va être alors évalué dans ce nouvel environnement et la valeur de `fib` sera trouvée dans a.

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

Correction : Oui. En effet,

```
(define f (lambda (...) (... f ... g ...)))
(define g (lambda (...) (... f ... g ...)))
```

va ajouter dans l'environnement courant k deux définitions pour `f` et `g` qui font toutes les deux référence à ce même environnement k (comme expliqué à la question précédente pour `fib`). Une fois `f` appliquée, elle peut donc trouver aussi bien la définition de `f` que celle dans `g` dans son environnement.

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.

Correction : Attention à ne pas oublier `quote` autour de la liste vide.

```
(define map (lambda (f l) (if l (cons (f (car l)) (map f (cdr l))) (quote ()))))
(define range (lambda (a b) (if (= a b) (quote ()) (cons a (range (+ a 1) b)))))
```

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, $\dots$, `Cons (ref` e_n, `Nil)` $\dots$ `))`.

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

Correction : Pas de difficulté :

```
let cons x y = Cons (ref x, y)
let car = function Cons (e, _) -> !e | _ -> invalid_arg "car"
let cdr = function Cons (_, e) ->  e | _ -> invalid_arg "cdr"
```

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).

Correction : On donne ici la totalité de la fonction `eval`, c'est-à-dire plus que ce qui était demandé. C'est assez long, alors il faut s'organiser soigneusement. On commence par de petites fonctions de construction et d'accès :

```
let list l = List.fold_right cons l Nil
let first2 e = car e, car (cdr e)
let first3 e = car e, car (cdr e), car (cdr (cdr e))
```

Puis on se donne des fonctions pour chercher dans l'environnement, correspondant à *assoc* et *lookup* :

```
let rec assoc x = function
  | Nil -> Nil
  | Cons ({contents = Cons ({ contents = Sym y }, _) as p}, tail) ->
      if x = y then p else assoc x tail
let rec lookup x = function
  | Nil -> Nil
  | Cons (head, tail) ->
    let p = assoc x !head in if p = Nil then lookup x tail else p
```

la fonction suivante pour ajouter dans l'environnement :

```
let define x v = function
  | Cons (head, _) -> head := cons (list [x; v]) !head
```

et enfin la fonction suivante pour construite la liste d'association entre les paramètres formels et les paramètres effectifs :

```
let rec make_assoc params args = match params, args with
  | Nil, Nil -> Nil
  | Cons (x, params), Cons (v, args) ->
      cons (list [!x; !v]) (make_assoc params args)
```

On peut enfin écrire la fonction `eval`

```
let rec eval env = function
  | Num _ as e ->
      e
  | Sym x ->
      car (cdr (lookup x env))
  | Cons (e1, e2) ->
      match !e1 with
      | Sym "quote" ->
          car e2
      | Sym "if" ->
          let test, e1, e2 = first3 e2 in
          eval env (if eval env test = Nil then e2 else e1)
      | Sym "define" ->
          let x, e = first2 e2 in
          define x (eval env e) env;
          Nil
      | Sym "lambda" ->
          let params, body = first2 e2 in
```

```
          list [params; body; env]
      | Sym "begin" ->
          sequence env e2
      (* primitives omises ... *)
      | head -> (* appel *)
          let head = eval env head in
          let args = eval_list env e2 in
          let params, body, env = first3 head in
          eval (cons (make_assoc params args) env) body
```

récursivement avec une fonction `eval_list` qui évalue toute une liste d'expressions

```
and eval_list env = function
  | Nil -> Nil
  | Cons (head, tail) -> cons (eval env !head) (eval_list env tail)
```

et une fonction `sequence` qui correspond à `begin`

```
and sequence env = function
  | Cons (head, Nil ) -> eval env !head
  | Cons (head, tail) -> ignore (eval env !head); sequence env tail
  | e                 -> eval env e
```

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.

Correction : La liste vide est représentée par 0. Toute autre expression est représentée par un pointeur vers un bloc (sur le tas), dont le premier mot identifie la nature ; par exemple | 1 | n | pour un entier, | 2 | adr. chaîne | pour un symbole et | 3 | *car* | *cdr* | pour un cons.

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

Correction :

```
car:
        movq    8(%rdi), %rax
        ret
cdr:
        movq    16(%rdi), %rax
        ret
cons:
        movq    %rdi, %rbx
        movq    %rsi, %r12
```

```
        movq    $24, %rdi
        call    malloc
        movq    $3, 0(%rax)
        movq    %rbx, 8(%rax)
        movq    %r12, 16(%rax)
        ret
```

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`.

Correction : On suppose qu'on vient de vérifier que
– `%rdi` est un cons ;
– son premier élément est le symbole `if`.
On suppose par ailleurs que la fonction `eval` ne modifie pas `%rsi` (l'environnement).

```
eval_if:
        movq    16(%rdi), %rdi
        pushq   16(%rdi)        # push (e2 e3)
        movq    8(%rdi), %rdi
        call    eval            # eval e1
        testq   %rax, %rax      # nil ?
        jz      eval_if_else
        popq    %rdi
        movq    8(%rdi), %rdi
        jmp     eval            # eval e2
eval_if_else:
        popq    %rdi
        movq    16(%rdi), %rdi
        movq    8(%rdi), %rdi
        jmp     eval            # eval e3
```

Noter l'appel terminal à `eval` pour évaluer e_2 ou e_3.

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.

Correction : Il est nécessaire d'ajouter quelques nouvelles primitives :

– une primitive `num?` pour déterminer si une valeur est un entier ;
– une primitive `sym?` pour déterminer si une valeur est un symbole ;
– une primitive `equal?` pour comparer entre eux deux symboles ;
– une primitive `set-car!` pour modifier le premier élément d'une liste.
On peut alors écrire la fonction `eval` :

```
(define eval
  (lambda (e k)
    (if (num? e)
        e
    (if (sym? e)
        (lookup e k)
    (begin
    (define head (car e))
    ;; (quote e1)
    (if (equal? head (quote quote))
        (car (cdr e))
    ;; (if e1 e2 e3)
    (if (equal? head (quote if))
        (if (eval (car (cdr e)) k)
            (eval (car (cdr (cdr e))) k)
            (eval (car (cdr (cdr (cdr e)))) k))
    ...
```

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

Correction : Attention : ne pas oublier `quote` et le fait que l'environnement est une liste contenant une liste d'association vide.

```
(eval (quote (+ 40 2)) (quote (())))
```

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`.