Devoir sur table – Novembre 2005
LI101
Durée : 1h30

Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le barème (total sur 40) apparaissant dans chaque boîte n'est donné qu'à titre indicatif.

La clarté des réponses et la présentation des programmes seront appréciées. Ne pas désagrafer les feuilles.

Exercice 1

Question 1.1 – Quels sont les résultats d'évaluation des expressions suivantes :

```
(let ((x 3))
  (let ((x 5)
        (y (+ x 7)))
    (* x y)))
```

[1/40]

```
(let* ((x 3))
  (let* ((x 5)
         (y (+ x 7)))
    (* x y)))
```

[1/40]

```
(+ (if (> 2 3) (/ 5 0) 4)
   (if (and (< 3 4) (not (= 4 6))) (* 3 5) 6)
   (if (or (> 6 2) (= 3 (/ 7 0))) 12 1))
```

[2/40]

Section	Groupe	Nom	Prénom

Exercice 2

Question 2.1 – Spécifier et définir la fonction nommée `au-moins-2?` qui teste si une liste passée en paramètre contient au moins 2 éléments :

```
(au-moins-2? (list 4 3 2))  → #T
(au-moins-2? (list 4))  → #F
(au-moins-2? (list "a" "little" "rabbit"))  → #T
```

[2/40]

Question 2.2 – Spécifier et définir une fonction nommée `produit-liste` telle que `(produit-liste L)` renvoie le produit des éléments d'une liste. Ainsi,

```
(produit-liste (list 1 2 3 4))  → 24
(produit-liste (list 2 9 283 0 23))  → 0
(produit-liste (list))  → 1
```

[2/40]

Section	Groupe	Nom	Prénom

Question 2.3 – Soit la fonction `mystere` suivante :

```
(define (mystere x y)
   (if (pair? y)
       (cons (* x (car y)) (mystere x (cdr y)))
       (list)))
```

Evaluer en pas-à-pas :

`(mystere 4 (list 2 3 4))`

En déduire une spécification pour la fonction `mystere` :

[3/40]

Question 2.4 – Soit le semi-prédicat `a-la-position` dont la spécification est la suivante :

;;; *a-la-position: nat * LISTE[alpha] -> α + #f*
;;; *(a-la-position n L) retourne l'élément à la position n dans la liste L*
;;; *s'il n'y a pas de n-ième position, la valeur retournée est #f*

Voici quelques exemples d'utilisation :

```
(a-la-position 2 (list 3 4 5 7 13 29))  → 5
(a-la-position 3 (list "u" "bu" "cucu" "dududu"))  → "dududu"
(a-la-position 2 (list 1 2))  → #F
(a-la-position 0 (list 1 2))  → 1
(a-la-position 2 (list))  → #F
```

Donner une définition en Scheme de cette fonction :

[4/40]

Section	Groupe	Nom	Prénom

Exercice 3

Whilhelm Ackermann (1896-1962) est un mathématicien allemand connu pour ses travaux en logique formelle. Mais on le connaît aujourd'hui surtout pour la découverte d'une fonction qui est une petite révolution à elle toute seule[1].

Soit la fonction d'Ackermann $ack(n,p)$ avec n et p entiers ≥ 0 définie par :

$$ack(n,p) = \begin{cases} p+1 & \text{si}\, n = 0 \\ ack(n-1,1) & \text{si}\, n > 0 \text{ et}\, p = 0 \\ ack(n-1, ack(n,p-1)) & \text{sinon} \end{cases}$$

Bien que cela soit non trivial, on peut démontrer formellement dans l'algorithme d'Ackermann que la récursion est bien fondée et que pour toute valeur de n et de p, la récursion termine bien en un nombre fini d'étapes.

Question 3.1 – Voici quelques extraits de résultats fournis par la version Scheme de la fonction `ack` :

```
(ack 0 0)  → 1
(ack 0 1)  → 2
(ack 0 2)  → 3
(ack 0 12)  → 13
(ack 1 0) →  (ack 0 1)  → 2
(ack 2 0) → (ack 1 1) → (ack 0 (ack 1 0)) →  (ack 0 2)  → 3
...
(ack 4 2) → (ack 3 (ack 4 1))
          → (ack 3 (ack 3 (ack 4 0)))
          → (ack 3 (ack 3 (ack 3 1)))
          → (ack 3 (ack 3 (ack 2 (ack 3 0)))
          → etc...
```

Donner la signature et définir en Scheme la fonction d'Ackermann `ack` :

[4/40]

La fonction d'Ackermann, malgré la simplicité des calculs effectués, possède la fâcheuse caractéristique d'être extrêmement lente à calculer. On peut montrer, en effet, qu'il faut un nombre exponentiel d'appels récursifs en fonction de n et p pour obtenir le résultat. Ainsi, le calcul de `(ack 4 1)` prend de longues minutes et `(ack 4 2)` ne peut être calculé en un temps raisonnable (disons moins de quelques millions d'années) même si l'on prenait l'intégralité des ordinateurs de la planète et qu'on leur demandait de faire ensemble ce calcul !

[1] Voir les définitions de `Whilhelm Ackermann` et `Ackermann function` sur l'encyclopédie Wikipedia à l'adresse `http://www.wikipedia.org/`

Section	Groupe	Nom	Prénom

Question 3.2 – Soit la fonction `memo-simple-aux` telle que `(memo-simple-aux n p pmax)` construit la liste des valeurs successives de `(ack n j)` où `j` varie de `p` à `pmax` par pas de 1.

Par exemple :

```
(memo-simple-aux 2 1 4)
→  (list (ack 2 1) (ack 2 2) (ack 2 3) (ack 2 4))  → (5 7 9 11)
```

Spécifier et définir la fonction `memo-simple-aux`. Puis en déduire une définition (sans spécification) de la fonction `memo-simple` telle que `(memo-simple n pmax)` retourne la liste des valeurs successives de `(ack n j)` où `j` varie de `0` à `pmax` par pas de 1 telle que :

```
(memo-simple 2 4)  → (3 5 7 9 11)
```

[4/40]

Question 3.3 – Soit la définition suivante :

```
;;; recherche-memo-simple: nat * LISTE[nat] -> nat + #f
(define (recherche-memo-simple p memo)
   (a-la-position p memo))
```

Donner le résultat d'évaluation de l'expression suivante :

```
(recherche-memo-simple 3 (memo-simple 2 4))
```

[2/40]

Section	Groupe	Nom	Prénom

Question 3.4 – Soit la fonction `memo` telle que `(memo nmax pmax)` retourne une liste dont l'élément à la position `i` (avec `i` variant de `0` à `nmax` par pas de 1) est une liste des valeurs successives de `(ack i j)` où `j` varie de `0` à `pmax`.

Par exemple :

```
(memo 3 2) → (list (list (ack 0 0) (ack 0 1) (ack 0 2))
                   (list (ack 1 0) (ack 1 1) (ack 1 2))
                   (list (ack 2 0) (ack 2 1) (ack 2 2))
                   (list (ack 3 0) (ack 3 1) (ack 3 2)))
→ ((1 2 3)
   (2 3 4)
   (3 5 7)
   (5 13 29))
```

Donner la signature et définir en Scheme la fonction `memo` :

[5/40]

Question 3.5 – Soit la définition suivante :

```
;;; recherche-memo: nat * nat * LISTE[LISTE[nat]] -> nat + #f
(define (recherche-memo n p memo)
   (let ((trouve (a-la-position n memo)))
     (if trouve
         (recherche-memo-simple p trouve)
         #f)))
```

Donner le résultat d'évaluation de l'expression suivante :
`(recherche-memo 2 3 (memo 3 4))`

[2/40]

Section	Groupe	Nom	Prénom

Question 3.6 – On peut donner une nouvelle définition de la fonction d'Ackermann. Cette nouvelle fonction, nommée `ack-memo` utilise les résultats précalculés présents dans `memo`.

Compléter la définition suivante de `ack-memo` :

```
(define (ack-memo n p memo)
                                                    [2/40]
  (let ((trouve                                          ))
     (if trouve
         trouve
         (cond ((= n 0) (+ p 1))
               ((= p 0) (ack-memo (- n 1) 1 memo))
                                                    [2/40]
               (else                                     )))))
```

Cette fonction nous permet un calcul assez rapide de ack(3,9) en écrivant :

`(ack-memo 3 9 (memo 3 6))`

Question 3.7 – On suppose connue la fonction `puissance`, vue en cours, dont la spécification est la suivante :

;;; *puissance: Nombre * nat -> Nombre*
;;; *(puissance x n) retourne x élevé à la puissance n*

Soit la fonction nommée `puitour` telle que `(puitour x n)` est définie en Scheme de la façon suivante :

```
(define (puitour n p)
  (if (= p 1)
      n
      (puissance n (puitour n (- p 1)))))
```

Donner la signature de cette fonction :

[2/40]

En fait, cette fonction retourne ce que l'on nomme une tour d'exponentielles, par exemple :

`(puitour 2 2)` $\rightarrow$ `4` $\equiv 2^2$
`(puitour 2 3)` $\rightarrow$ `16` $\equiv 2^{2^2} \equiv 2^4$
`(puitour 2 4)` $\rightarrow$ `65536` $\equiv 2^{2^{2^2}} \equiv 2^{16}$
`(puitour 2 5)` $\rightarrow 2^{2^{2^{2^2}}} = 2^{65536} = 2003529$ *...etc...*

Section	Groupe	Nom	Prénom

Question 3.8 – On se propose de calculer `(ack 4 2)` en un temps record

Pour cela, on utilise l'identité remarquable suivante :

$$ack(4,p) = -3 + \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} \text{tour de puissances de } p+3 \text{ étages}$$

En utilisant cette identité remarquable, spécifier et définir une fonction unaire nommée `ack-4` effectuant un calcul rapide de $ack(4,p)$

[2/40]

Ceci nous permet de calculer rapidement $ack(4,2)$ de la façon suivante :

`(ack-4 2)` → *2003529930406846464979 0 ... etc ...*

Il faut en fait 19729 chiffres pour écrire le résultat de ce calcul (tester à la maison). Cette méthode possède quelques limites car elle ne marche plus à partir de `(ack-4 3)`

En effet, l'écriture seule du résultat de ce calcul nécessite un nombre de chiffres plus grand que le nombre estimé d'atomes dans l'univers visible ! Certains se seraient laissés tenter ...