Examen – Module d'informatique 1 – Janvier 2000
MIAS 1ère année – 1er semestre

Aucun document ni machine électronique n'est permis.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue (utiliser le dos de la feuille précédente si la réponse déborde des boîtes). Le barème (sur 50) apparaissant dans chaque boîte n'est donné qu'à titre indicatif.

La clarté des réponses et la présentation des programmes seront appréciées. Toutes les fonctions apparaissant dans les réponses doivent être accompagnées de leur spécification. Ne pas désagrapher les feuilles.

Exercice 1

Écrire une fonction, nommée `cube`, calculant le volume d'un cube d'arête a.

Réponse [3/50]

```
;;; Retourne le volume d'un cube d'arête a.
;;;    Nombre -> Nombre
(define (cube a)
  (* a a a) )
```

Commentaire: Il est plus lisible d'utiliser les capacités n-aires de la multiplication et donc d'écrire `(* a a a)` plutôt que `(* a (* a a))`.

Exercice 2

DANS CET EXERCICE, ON SUPPOSE NE PAS AVOIR D'ENTIERS EN SCHEME. On représentera les entiers naturels par des *EntierBaton*s c'est-à-dire des listes de bâtons. Ainsi 0 sera-t'il représenté par la liste vide, 1 par la liste `(I)`, 2 par la liste `(I I)` et ainsi de suite.

Question 2.1 – Écrire un prédicat, nommé `positif?`, testant si un *EntierBaton* est strictement positif.

Réponse [3/50]

```
;;; Teste si un entier est strictement positif.
;;;    EntierBaton -> booléen
(define (positif? p)
  (pair? p) )
```

Commentaire: La spécification stipule que l'argument est un *EntierBaton*, il n'est donc nul besoin de vérifier que c'est une liste de bâtons (un bâton étant représenté par le symbole `I`) ni d'inventer une représentation pour des entiers négatifs puisque l'on ne s'occupe que d'entiers naturels.

Question 2.2 – Écrire une fonction, nommée `plus`, additionnant deux *EntierBaton*s et retournant leur somme sous la forme d'un *EntierBaton*. Par exemple,

`(plus '(I I I) '(I I))` → `(I I I I I)`

Section | Numéro d'anonymat

Réponse [4/50]

```
;;; Additionne deux entiers représentés par des listes de bâtons
;;;    EntierBaton * EntierBaton -> EntierBaton
(define (plus p q)
  (append p q) )

(define (plus p q)
  (if (positif? p)
      (cons (car p) (plus (cdr p) q))
      q ) )
```

Commentaire: La seconde solution utilise le classique schéma de récursion sur les listes. La première tire parti de la représentation et emploie l'utilitaire `append` bien connu.

Question 2.3 – Écrire une fonction, nommée `moins1`, qui prend un *EntierBaton* strictement positif et lui retranche l'*EntierBaton* correspondant au nombre 1.

Réponse [3/50]

```
;;; Retranche un d'un entier plus grand ou égal à un.
;;;    EntierBaton[>= 1] -> EntierBaton
(define (moins1 p)
  (cdr p) )
```

Commentaire: On peut aussi utiliser le schéma récursif en arrêtant la récursion non pas à zéro mais à un.

```
(define (moins1rec p)
  (if (and (pair? p)              ; ou (equal? p '(I))
           (null? (cdr p)) )
      '()
      (cons (car p) (moins1rec (cdr p))) ) )
```

Question 2.4 – Écrire une fonction, nommée `fois`, multipliant deux *EntierBaton*s et retournant leur produit sous la forme d'un *EntierBaton*. On pourra utiliser la relation suivante :

$$\text{pour } p >= 1,\ pq = q + (p-1)q$$

Section | Numéro d'anonymat

Réponse [5/50]

```
;;; Multiplie deux entiers représentés par des listes de bâtons
;;;    EntierBaton * EntierBaton -> EntierBaton
(define (fois p q)
  (if (positif? p)
      (plus q (fois (moins1 p) q))
      (zero) ) )

;;; Retourne l'entier bâton zéro.
;;;     void -> EntierBaton
(define (zero)
  '() )
```

Commentaire: Le cas d'arrêt n'était pas mentionné dans l'énoncé. On peut donc ajouter la règle :

$$\text{pour } p = 0,\ pq = 0$$

Observer que l'on emploie les fonctions précédentes `positif?`, `plus`, `moins1` et `zero` qui forment une nouvelle barrière d'abstraction qui permettent à la fonction `fois` d'ignorer la représentation exacte des entiers bâtons.

Exercice 3

Le but de cet exercice est de définir un prédicat, nommé `equilibre?`, qui prend en arguments un prédicat p et une liste l. Le prédicat `equilibre?` retourne vrai s'il y a, dans la liste l, autant de termes qui satisfont p que de termes qui ne le satisfont pas. Par exemple,

```
(equilibre? zero?  '(1 0 0 5 4))    → #f
(equilibre? number? '(1 a c 3 e 5)) → #t
```

Question 3.1 – Dans un premier temps, on demande d'écrire une fonction, nommée `compter`, qui prend les mêmes arguments que `equilibre?` et qui retourne la différence entre le nombre de termes de la liste l qui satisfont le prédicat p et le nombre de termes de la liste l qui ne satisfont pas le prédicat p.

Section Numéro d'anonymat

Réponse [4/50]

```
;;; Calcule la différence entre le nombre de termes de la liste l qui
;;; satisfont le prédicat p et le nombre de ceux qui ne le satisfont
;;; pas.
;;;     (a -> bool) * a* -> int
(define (compter p l)
  (if (pair? l)
      (if (p (car l))
          (+ (compter p (cdr l)) 1)
          (- (compter p (cdr l)) 1) )
      0 ) )
```

MIAS11 connaissait la fonction `filtrer` que l'on pouvait également utiliser :

```
;;; Filtrer une liste à l'aide d'un prédicat.
;;;      (a -> bool) * a* -> a*
(define (filtrer predicat? l)
  (if (pair? l)
      (let ((reste (filtrer predicat? (cdr l))))
        (if (predicat? (car l))
            (cons (car l) reste)
            reste ) )
      '() ) )

;;; Compter à l'aide de filtrer.
;;;     (a -> bool) * a* -> int
(define (compterf p l)
  (define (notp e)
    (not (p e)) )
  (- (filtrer p l) (filtrer notp l)) )
```

Question 3.2 – Définir maintenant le prédicat `equilibre?`.

Réponse [3/50]

```
;;; Teste si une liste est équilibrée vis à vis d'un prédicat p c'est-à-dire
;;; si elle possède autant de termes qui satisfont p que de termes qui ne le
;;; satisfont pas.
;;;    (a -> bool) * a* -> bool
(define (equilibre? p l)
  (= 0 (compter p l)) )
```

Exercice 4

Soit la propriété booléenne définie par le prédicat ternaire *coeurCarreau* définie sur les entiers naturels par les règles suivantes :

$$\begin{array}{lll} coeurCarreau(a,0,c) = & \text{Vrai} & \text{si } a = c \\ coeurCarreau(a,0,c) = & \text{Faux} & \text{si } a \neq c \\ coeurCarreau(a,b,c) = & coeurCarreau(a-b,b,c) & \text{si } a > b \\ coeurCarreau(a,b,c) = & coeurCarreau(a,b-a,c) & \text{si } a <= b \end{array}$$

Question 4.1 – Que vaut $coeurCarreau(24,8,4)$? Que vaut $coeurCarreau(24,16,8)$?

Réponse [2/50]
$coeurCarreau(24,8,4) = coeurCarreau(16,8,4) = coeurCarreau(8,8,4) = coeurCarreau(8,0,4) =$ Faux.
$coeurCarreau(24,16,8) = coeurCarreau(8,16,8) = coeurCarreau(8,8,8) = coeurCarreau(8,0,8) =$ Vrai.

Section | Numéro d'anonymat

Question 4.2 – Écrire un prédicat ternaire, que l'on invoquera comme `(coeur-carreau` a b c`)`, implantant en Scheme le calcul de l'expression $coeurCarreau(a, b, c)$.

Réponse [4/50]

```
;;; Teste si trois nombres a, b et c sont tels que c = pgcd(a,b).
;;;     Nat * Nat * Nat -> bool
(define (coeur-carreau a b c)
  (if (= b 0)
      (= a c)
      (if (> a b)
          (coeur-carreau (- a b) b c)
          (coeur-carreau a (- b a) c) ) ) )
```

Commentaire: Cette fonction teste que ses trois arguments a, b, c sont tels que $pgcd(a, b) = c$. La difficulté était d'ordonner les quatre règles définissant le calcul afin d'obtenir un arbre de décision. Les deux premières sont déclenchées lorsque la variable b vaut zéro et sont différenciées par la comparaison de a et b. Les deux dernières règles sont différenciées par la comparaison $a < b$. On évitera bien sûr d'écrire, sans les simplifier, des expressions telles que `(if` α `#t #f)`.

Exercice 5

Soit le petit langage, nommé CC, défini par la grammaire suivante (qui comporte quatre règles) :

<programme>	→	(*<programme>* `OU` *<programme>*)	Règle 1
		(*<expression>* `COEUR` *<expression>* `CARREAU` *<expression>*)	Règle 2
<expression>	→	*<naturel>*	Règle 3
		(*<expression>* `-` *<expression>*)	Règle 4

On rappelle qu'un entier naturel est un entier positif ou nul.

Question 5.1 – Donner deux exemples de S-expressions satisfaisant cette grammaire et utilisant au moins une fois les quatre règles qu'elle définit.

Réponse [2/50]

```
( 3 COEUR (3 - 1) CARREAU 1 )
(( (5 - 3) COEUR (3 - (2 - 1)) CARREAU 1 )
 OU ( 2 COEUR 2 CARREAU 2 ) )
```

Question 5.2 – Qu'est-ce qu'une barrière d'abstraction ?

Réponse [3/50]

Un ensemble de fonctions permettant de manipuler des concepts sans exposer leur représentation.

Dans le langage CC, la notation p `OU` q représente la disjonction (le « ou » des booléens) des valeurs de vérité p et q. Les expressions composées avec les symboles `COEUR` et `CARREAU` utilisent pour être évaluées, la fonction `coeur-carreau` de l'exercice précédent. Plus précisément, la valeur du programme (a `COEUR` b `CARREAU` c) est la valeur de $coeurCarreau(a, b, c)$. La soustraction est repérée par le symbole `-` en position infixe.

L'évaluateur du langage CC utilise une barrière d'abstraction syntaxique dont voici quelques éléments. On suppose que programmes et expressions sont syntaxiquement corrects.

```
;;; Programme[correct] -> bool
(define (disjonction? p)
  (equal? (cadr p) 'OU) )
```

Section | Numéro d'anonymat

```
;;; Expression[correcte] -> bool
(define (nombre? e)
  (number? e) )

;;; Expression[nombre] -> int
(define (nombre-valeur n)
  n )

;;; Programme[coeur-carreau] -> Expression
(define (coeur-carreau-operande-1 p)
  (car p) )

;;; Programme[coeur-carreau] -> Expression
(define (coeur-carreau-operande-2 p)
  (car (cddr p)) )

;;; Programme[coeur-carreau] -> Expression
(define (coeur-carreau-operande-3 p)
  (car (cddr (cddr p))) )
```

Question 5.3 – Définir les quatre sélecteurs manquant dans la précédente barrière d'abstraction syntaxique.

Réponse [2/50]

```
;;; Programme[disjonction] -> Programme
(define (disjonction-operande-1 p)
  (car p) )

;;; Programme[disjonction] -> Programme
(define (disjonction-operande-2 p)
  (caddr p) )

;;; Expression[différence] -> Expression
(define (difference-operande-1 e)
  (car e) )

;;; Expression[différence] -> Expression
(define (difference-operande-2 e)
  (caddr e) )
```

Question 5.4 – Compléter les définitions suivantes.

```
;;; Évalue un programme
;;;   Programme -> booléen
(define (programme-eval p)
  (if (disjonction? p)
      cas de la règle 1
      cas de la règle 2 ) )

;;; Évaluer une expression
;;;   Expression -> entier
(define (expression-eval e)
  (if (nombre? e)
      cas de la règle 3
      cas de la règle 4 ) )
```

Section Numéro d'anonymat

Réponse [7/50]

```
;;; Évalue un programme
;;;   Programme -> booléen
(define (programme-eval p)
  (if (disjonction? p)
      (or (programme-eval (disjonction-operande-1 p))
          (programme-eval (disjonction-operande-2 p)) )
      (coeur-carreau (expression-eval (coeur-carreau-operande-1 p))
                     (expression-eval (coeur-carreau-operande-2 p))
                     (expression-eval (coeur-carreau-operande-3 p)) ) ) )

;;; Évaluer une expression
;;;   Expression -> entier
(define (expression-eval e)
  (if (nombre? e)
      (nombre-valeur e)
      (- (expression-eval (difference-operande-1 e))
         (expression-eval (difference-operande-2 e)) ) ) )
```

Exercice 6

Examiner la fonction suivante :

```
(define (mystere o)
  (define (secret o)
    (if (pair? (cdr o))
        (secret (cdr o))
        (mystere (car o)) ) )
  (if (pair? o)
      (secret o)
      o ) )
```

Question 6.1 – Évaluer à la main l'expression `(mystere '((a c e) f (b d)))`. Quelles sont les (environ) sept étapes principales de cette évaluation ?

Réponse [2/50]

Voici les étapes principales :

```
(mystere '((a c e) f (b d)))
≡ (secret '((a c e) f (b d)))
≡ (secret '(f (b d))
≡ (secret '((b d))
≡ (mystere '(b d))
≡ (secret '(b d))
≡ (secret '(d))
≡ (mystere 'd)
→ d
```

Question 6.2 – Donner une spécification aux fonctions `mystere` et `secret`.

Réponse [3/50]

La fonction `mystere` retourne le terme atomique le plus à droite d'une S-expression quelconque. La fonction `secret` fait la même chose mais suppose, en outre, que son argument est une S-expression satisfaisant le prédicat `pair ?`.

