Université Nice Sophia-Antipolis
L2 Info
Programmation Fonctionnelle Avancée – session 1
Examen : 17 décembre 2018
Durée : 1 heure 30

Note

Prénom _____

Nom _____

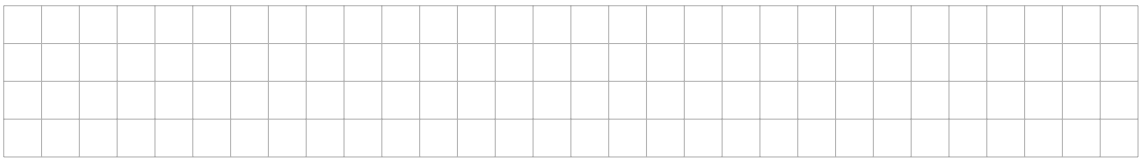
Num. Etu.

Tous les documents sont interdits. Les calculatrices et les téléphones portables doivent être rangés ainsi que tout autre matériel à l'exception d'un stylo et d'un effaceur. Un code mal parenthésé sera considéré comme faux s'il est de plus mal indenté.

Exercice 1 Somme vectorielle (5 points)

En utilisant une définition par récurrence, définissez une fonction `(somme-vectorielle v1 v2)` qui prend en argument deux vecteurs `v1` et `v2` représentés par des listes, et qui renvoie leur somme vectorielle. On suppose que `v1` et `v2` sont des vecteurs de même dimension, autrement dit les listes sont de même longueur, mais la dimension peut être arbitraire. Par exemple,
`(somme-vectorielle '(1 2 3) '(4 5 6))` renvoie `(5 7 9)` et
`(somme-vectorielle '(1 2 3 4) '(5 6 7 8))` renvoie `(6 8 10 12)`.

En utilisant la fonction d'ordre supérieur `map`, reprogrammez cette fonction en une ligne, sans utiliser d'appel par récurrence.



.

Reprogrammez la fonction `somme-vectorielle` en utilisant les fonctions `list-ref` et `build-list`, et en vous aidant du code ci-dessous.

```
(define (somme-vectorielle vect1 vect2)
  (build-list ...
    (lambda (i) (+ (list-ref ... ) (list-ref ... )))))
```

On rappele que `(build-list n f)` renvoie la liste `((f 0) ... (f (- n 1)))` et que `(list-ref (a0 ... an) i)` renvoie `ai`.

Généralisez votre fonction pour qu'elle puisse prendre $n > 0$ vecteurs en argument, toujours tous de même dimension. Par exemple,
`(somme-vectorielle '(1 2 3) '(4 5 6) '(7 8 9))` renvoie `(12 15 18)` et
`(somme-vectorielle '(1) '(2) '(3) '(4))` renvoie `(10)`.

Exercice 2 Lecture de partition (4 points)

On suppose que l'on dispose d'une fonction `(play-tone tone delay)` qui prend en argument le nom d'une note et une durée en secondes et qui joue cette note la durée demandée, et retourne `void` « immédiatement », autrement dit au moment où la note est sur le point d'être jouée.

Par exemple,
`(begin (play-tone 'do 1) (play-tone 'mi 1) (play-tone 'sol 1))`
joue l'accord do-mi-sol durant une seconde (les trois notes sont jouées en même temps), tandis que
`(begin (play-tone 'do 1) (sleep 1) (play-tone 'mi 2))`
joue un do durant 1s, puis un mi durant 2s, sans chevauchement.

Écrivez une fonction `(play-score L)` qui prend en argument une liste de couples `(tone delay)` et qui joue chaque note dans l'ordre avec la durée indiquée. Par exemple `(play-score '((do 1) (mi 2)))` joue un do durant 1s puis un mi durant 2s, sans chevauchement.

.

A quel moment votre fonction `(play-score L)` retourne-t-elle `void` ?

On suppose maintenant que la fonction `play-tone` peut prendre un troisième argument optionnel indiquant le temps à attendre avant de commencer à jouer la note. Par exemple, `(play-tone 'mi 2 1)` donne l'ordre de jouer un mi d'une durée de 2s en attendant 1s « à partir de maintenant » avant de commencer à le jouer. La fonction retourne « immédiatement ». Par exemple,
`(begin (play-tone 'do 1) (play-tone 'mi 2 1))`
joue un do durant 1s, puis un mi durant 2s, sans chevauchement.

Écrivez une nouvelle version de la fonction `(play-score L)` qui retourne immédiatement.

.

Exercice 3 Gestion d'erreur (3 points)

On considère la fonction

```
(define (if-error p q) (when (equal? p 'error) q))
```

Que s'affiche-t-il si on évalue `(if-error (printf "hello") (printf "world"))` ?

À l'aide de `define-macro`, proposez une autre définition de `if-error` telle que si l'évaluation de `p` renvoie `'error`, on évalue `q`, et sinon `q` n'est pas évalué. Par exemple `(if-error (printf "hello") (printf "world"))` affiche `hello`, mais
`(if-error (begin (printf "hello") 'error) (printf "world"))`
affiche `hello world`.

Rappel : voici un exemple de définition de macro

```
(define-macro si
  (syntax-rules (alors sinon)
    ((si test alors p sinon q) (if test p q))))
```

Exercice 4 Listes à la Python (5 points)

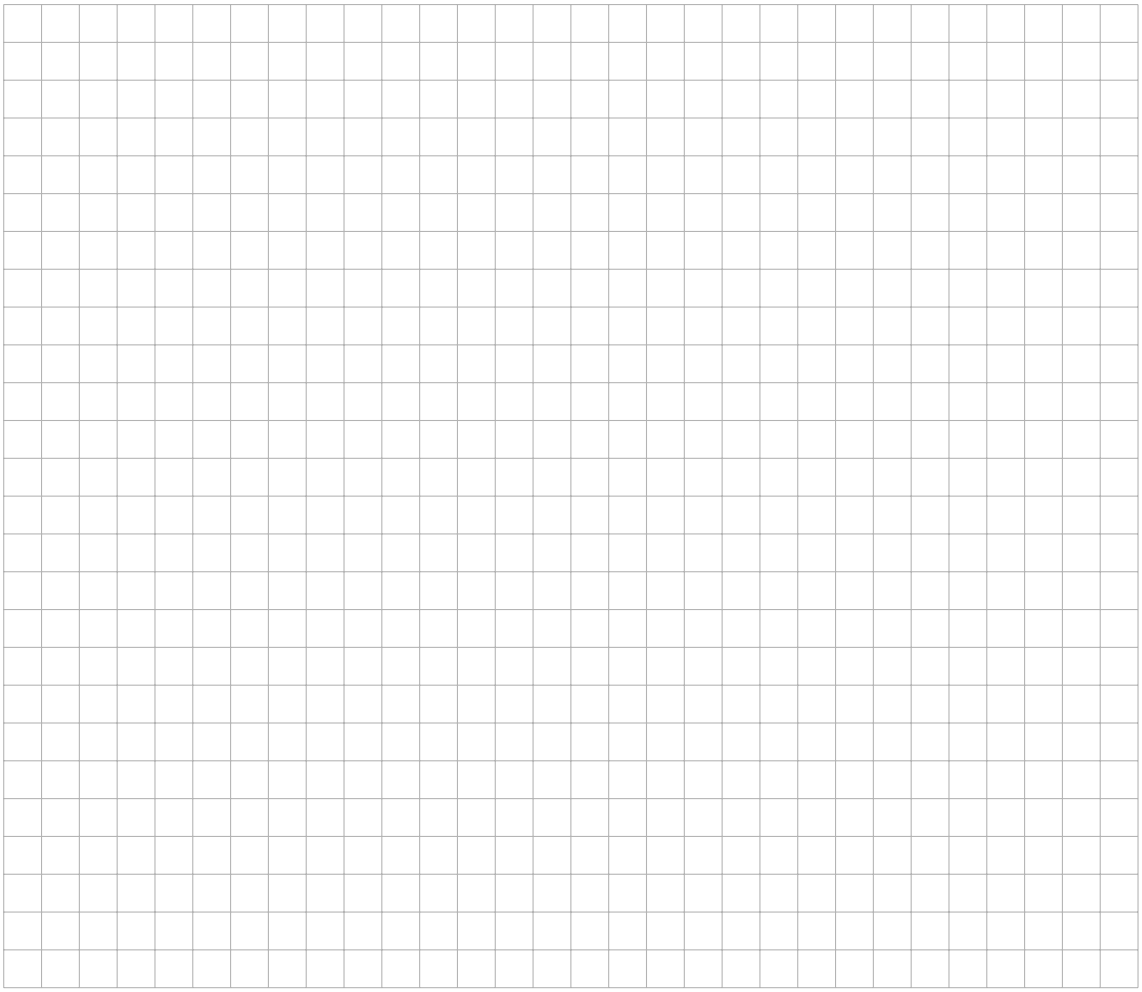
On souhaite ajouter à Scheme des listes à la Python, autrement dit des listes avec une méthode `append` permettant de rajouter un élément en fin de liste. Pour cela, on va représenter la liste par un tableau Scheme dont on utilise seulement les premières cases. Lorsqu'on étend la liste avec `append`, on prend la première case non utilisée, si elle existe. Si toutes les cases du tableau sont utilisées, on commence par créer un nouveau tableau deux fois plus long. [1]
Complétez la définition de classe `pylist%` à la page suivante.
Rappels sur les tableaux en Scheme :

- `(make-vector n)` renvoie un nouveau tableau de longueur n
- `(vector-ref t i)` renvoie le contenu de la $i$ème case du tableau `t`
- `(vector-set! t i v)` remplace le contenu de la $i$ème case par `v`

1. En doublant la taille du tableau, on parvient à « amortir » le coût de la recopie du tableau ; en moyenne, append a une complexité en temps constant.

.

```
(define pylist%
  (class object%

    (init-field
     (vect (make-vector 1)) ;; le tableau, initialement de longueur 1
     (len 0))               ;; le nombre de cases utilisees, initialement 0

    (define/public (ref i)
      ;; L[i] en Python
      ;; renvoie le ieme element, ou error "out of bound"
      (if (and (>= i 0) (< i len)) (vector-ref vect i) (error "out␣of␣bound")))

    (define/public (set i v)
      ;; L[i] = v en Python
      ;; modifie le ieme element, ou error "out of bound"
    ...)

    (define/public (append v)
      ;; L.append(v) en Python
      ;; ajoute v en fin de liste
    ...)

    (define/public (pop)
      ;; L.pop() en Python
      ;; retire et renvoie le dernier element
      ;; ou error "pop from empty list"
    ...)

    (super-new)

    ))
```



.

Exercice 5 Générateurs (3 points)

On rappelle qu'un générateur est une fonction sans argument qui renvoie une valeur possiblement différente à chaque appel. Par exemple, le code ci-dessous est un générateur qui renvoie 0, puis 1, puis 2, etc.

```
(define next-int ;; un generateur pour 0,1,2,3,...
  (let [(c -1)]  ;; variable privee  de la cloture
     (lambda ()  ;; fonction sans argument!
        (set! c (+ c 1))
        c)))

(list (next-int) (next-int) (next-int))  ;; renvoie (0 1 2)
```

Définissez un générateur `next-power2` pour la suite des puissances de 2 $(1, 2, 4, 8, \ldots)$.

Définissez un générateur `next-fib` pour la suite de Fibonnacci $(1, 1, 2, 3, 5, 8, \ldots)$.

.

Définissez un générateur pour la suite des nombres premiers $(2, 3, 5, 7, 11, \ldots)$.

Indication : vous pouvez par exemple compléter le code ci-dessous.

```
(define next-prime
 (let [(prime? (lambda (x) #t))
       (n 2)]
  (lambda ()
   (if (prime? n)
       ;; if true
       (let [(res n)
             (not-divided-by-previous-primes? prime?)]
       (set! n (+ n 1))
       (set! prime?
         (lambda (x) (and (not-divided-by-previous-primes? x)
                          ... )))
       res)
       ;; if false
       (begin
           (set! n (+ n 1))
           (next-prime))))))
```