Conservatoire National des Arts et Métiers
292 rue Saint-Martin – 75141 PARIS Cedex 03
–0–

Équipe pédagogique « Statistique et Bioinformatique »

Corrigé de l'examen du 4 février 2016

TITRE DE L'ENSEIGNEMENT : Initiation à la programmation **(BNF102)**

Nature de l'enseignement : Cours
Cycle : **B**
Indice : **B**
Nombre de pages : 5
Nom du responsable : **Pr. Jean-François ZAGURY**
Année universitaire : **2015/2016 – 1ère session**
Documents autorisés : aucun

* * *

Les deux premières questions sont notées sur 7 points, la dernière sur 6 points.

Problème N° 1

Le triangle de Pascal, du nom de son inventeur[1], est un algorithme ingénieux qui permet de déterminer les coefficients du binôme, jusqu'à une certaine valeur n, sans avoir à les calculer.

Les coefficients du binôme pour la puissance n sont les coefficients du polynôme résultat du développement de l'expression $(a+b)^n$. Ainsi, pour $n = 4$:

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

On a bien : $(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$

On démontre par récurrence que les coefficients de ce polynôme sont, pour chaque valeur de n (l'exposant du binôme) et pour chaque valeur de k (l'exposant de b pour la colonne k du triangle), selon le système de notation choisi :

$$\binom{n}{k} = C_n^k$$

La formule mathématique qui donne le résultat directement est donc :

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

or :

1. En fait Pascal avait des précurseurs persans, comme al-Karaji (953 - 1029) ou Omar Khayyam au XI^{e} siècle, marocain, comme Ibn al-Banna au XIII^{e} siècle, chinois, comme Jia Xian au XI^{e} siècle et d'autres Européens comme Peter Apian et Michael Stifel en Allemagne, Tartaglia en Italie et François Viète au Poitou. Mais Blaise Pascal fut le premier à publier un exposé complet des 19 propriétés remarquables de ce triangle.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Parmi les propriétés remarquables de ce tableau triangulaire de nombres on note les suivantes :

$$\binom{n}{n} = C_n^n = 1$$

$$\binom{n}{0} = C_n^0 = 1$$

d'où il résulte que le premier et le dernier coefficient de chaque ligne du triangle seront égaux à 1.

On en déduit l'algorithme de construction du triangle de Pascal :

```
1  Algo      : Pascal
2  Données   : n ;; pour (a+b)^n
3  Soit      : T tableau carré de n+1 lignes
4  pour i allant de 0 à n faire
5     T[i,0] ← 1 ;; première colonne à 1
6  fin pour
7  pour i allant de 1 à n faire
8     pour j allant de 1 à i
9        T[i,j] ← T[i-1,j-1] + T [i-1,j]
10    fin pour
11 fin pour
```

On vous donne les procédures utilitaires suivantes, que vous devrez utiliser pour écrire vos programmes :

– pour construire un tableau carré, réalisé comme un vecteur de n lignes, chacune constituée d'un vecteur de n valeurs égales à 0 :

```
1 (define (fabriquer-tableau-carre n)
2    (let ((le-tableau (make-vector n #f)))
3       (do ((i 0 (+ i 1)))
4           ((= i n) le-tableau)
5           (vector-set! le-tableau i (make-vector n 0)))))
```

– pour accéder à une ligne de ce tableau :

```
1 (define (extraire-ligne tableau num-ligne)
2    (vector-ref tableau num-ligne))
```

– pour accéder à une case de ce tableau :

```
1 (define (valeur-case tableau num-ligne num-col)
2    (vector-ref (extraire-ligne tableau num-ligne) num-col))
```

– pour placer une valeur dans une case du tableau :

```
1 (define (affecte-case! tableau num-ligne num-col val)
2    (vector-set! (extraire-ligne tableau num-ligne) num-col val))
```

– pour afficher le contenu du tableau, à l'exclusion des valeurs égales à 0 :

```
(define (affiche tableau)
   (let ((n (vector-length tableau)))
      (do ((i 0 (+ i 1)))
          ((= i n))
          (do ((j 0 (+ j 1)))
              ((= j n)
               (newline))
              (let ((v (valeur-case tableau i j)))
                 (if (not (zero? v))
                     (display v))
                 (display #\space))))))
```

Il vous est demandé d'écrire le programme Scheme qui affichera le triangle de Pascal, pour une certaine valeur de n.

Réponse :

```
(define (pascal n)
   (let ((T (fabriquer-tableau-carre (+ n 1))))
      (do ((i 0 (+ i 1))) ;; initialisation de la
          ((> i n))       ;; première colonne à 1
          (affecte-case! T i 0 1))
      (do ((i 1 (+ i 1))) ;; on commence à 1
            ((> i n))     ;; on s'arrête à n
         (do ((j 1 (+ j 1))) ;; on affecte à chaque
             ((> j i))  ;; case la somme des valeurs
             (affecte-case! T i j ;; des cases i-1,j
                (+ (valeur-case T (- i 1) (- j 1));; et i-1,j-1
                   (valeur-case T (- i 1) j)))))
      T))
```

Problème N° 2

Le programme doit maintenant être mis sous la forme d'un module compilable pour produire un fichier binaire exécutable qui sera invoqué de la façon suivante, à l'invite du shell, si le binaire se nomme par exemple `pascal` et si l'on veut par exemple aller jusqu'à $n = 8$ pour notre triangle :

```
./pascal 8
```

On rappelle :

– que la ligne de commande d'appel de ce programme est transmise au programme sous la forme d'une liste, dans notre cas cette liste sera donc :
(./pascal 8)

– que la forme générale de l'en-tête d'un tel module est :

(**module** <nom du module>
   (**main** <nom de la procédure initiale>))

```
(define (<nom de la procédure initiale> Args)
   (let ((n (cadr Args)))
      ...
```

Vous écrirez ce programme compilable ; pour les procédures déjà vues pour la question précédente vous n'écrirez sur votre copie que la première ligne de chacune, c'est-à-dire son prototype précédé de (**define** .

Réponse :

```
1  (module pascal-mono
2    (main init))
3
4  (define (init argv)
5    (let ((N (string->number (cadr argv))))
6      (affiche (pascal N))))
7
8  (define (pascal n)
```

Problème N° 3

Nous disposons désormais du programme suivant :

```
1  (module pascal
2    (main init))
3
4  (define (init argv)
5    (let ((N (string->number (cadr argv))))
6      (affiche (pascal N))))
7
8  (define (pascal n)
9    (let ((T (fabriquer-tableau-carre (+ n 1))))
10     (do ((i 0 (+ i 1))) ;; initialisation de la
11         ((> i n))       ;; première colonne à 1
12         (affecte-case! T i 0 1))
13     (do ((i 1 (+ i 1))) ;; on commence à 1
14         ((> i n))       ;; on s'arrête à n
15       (do ((j 1 (+ j 1)))
16           ... ;; à compléter ici
17       T))
18
19 (define (fabriquer-tableau-carre n)
20   ...
21
22 (define (extraire-ligne tableau num-ligne)
23   ...
24
25 (define (affecte-case! tableau num-ligne num-col val)
26   ...
27
28 (define (valeur-case tableau num-ligne num-col)
29   ...
```

```
30
31 (define (affiche tableau)
32    ...
```

Il vous est demandé de le diviser en deux modules :

- **utile**, qui comportera les cinq dernières procédures du listing ci-dessus et qui exportera les procédures nécessaires à **pascal** ;
- **pascal**, qui comportera les procédures **init** et **pascal**, et qui importera ce qu'exporte **utile**.

On rappelle que la syntaxe pour exporter, par exemple, la procédure **affiche** est :

```
1 (module utile
2    (export (affiche tableau))
3    ...
```

Réponse :

Fichier `pascal-multi.scm` :

```
1  (module pascal
2     (main init)
3     (import utile))
4
5  (define (init argv)
6     (let ((N (string->number (cadr argv))))
7        (affiche (pascal N))))
8
9  (define (pascal n)
10    (let ((T (fabriquer-tableau-carre (+ n 1)))))
```

Fichier `utilitaires.scm` :

```
1  (module utile
2     (export (fabriquer-tableau-carre n)
3             (affecte-case! tableau num-ligne num-col val)
4             (valeur-case tableau num-ligne num-col)
5             (affiche tableau)))
6
7  (define (fabriquer-tableau-carre n)
```

Il faut en outre fournir au compilateur le fichier `.afile` suivant :

```
1 ((pascal "pascal-multi.scm")
2  (utile  "utilitaires.scm"))
```