Université Pierre et Marie Curie
Formation permanente

– o –

# Prise en Main d'Objective Caml

Travaux encadrés sur machines - J2

Emmanuel Chailloux Pascal Manoury Bruno Pagano Michel Mauny

Février 2006

## 1 Deuxième jour : noyau impératif

### 1.1 Fusion de deux fichiers - version 1

1. Ecrire une fonction `fusion` de type `'a list -> 'a list -> 'a list` qui prend deux listes triées selon un ordre croissant et retourne la liste triée et croissante de l'union de ces deux listes.
2. Ecrire une fonction `lire` qui lit un fichier texte et retourne la liste des lignes de ce fichier.
3. Ecrire une fonction qui convertit une liste de chaînes de caractères représentant des nombres entiers en une liste d'entiers.
4. Créer la fonction principale `fusion_fichier` qui prend comme arguments trois noms de fichier et fusionne les deux premiers sur le 3ème en utilisant les fonctions précédentes.

### 1.2 Fusion de deux fichiers - version 2

On cherche à créer un nouveau fichier d'entiers triés à partir de deux fichiers d'entiers triés en ordre croissant.
**Exemple**

```
fichier1 : 3 7 9
fichier2 : 2 4 6 7 8 10

fichier3 : 2 3 4 6 7 7 8 9 10
```

Les fichiers sont constitués d'un entier (sous forme d'une chaîne) par ligne.

1. Créer 2 fichiers d'entiers triés, selon un ordre croissant, sous un éditeur de texte. On écrira un nombre par ligne.
2. Ecrire une fonction `lire_suivant` qui prend un `in_channel`, lit une ligne, la convertit en entier et retourne cette valeur.

3. Ecrire une fonction `fusion_channel` qui prend comme arguments 2 `in_channel` et 1 `out_channel` et effectue la fusion des entrées provenant des `in_channel` et l'écrit dans la sortie (`out_channel`).
4. Créer la fonction principale `fusion_fichier` qui prend comme arguments 3 noms de fichier et fusionne les 2 premiers sur le 3ème.

### 1.3 Piles

On veut implanter un type `'a stack` et les primitives de gestion classique des piles.

On représente les piles comme une référence sur une liste paramétrée :

```
type 'a stack = ('a list) ref;;
```

Les primitives modifieront physiquement la pile, si besoin.
On utilisera la coercion de type `(e : t)` pour forcer le type des différentes primitives en `'a stack`.

1. Ecrire une fonction `new_stack` de type `unit -> 'a stack` qui crée une nouvelle pile.
2. Ecrire une fonction booléenne `empty_stack` qui teste si la pile est vide.
3. Ecrire les primitives `push`, `top` et `pop`. Ces deux dernières fonctions déclencheront l'exception `Empty_stack` lorsqu'appliquées à une pile vide.
4. Définir un nouveau type `'a stack` en utilisant un enregistrement contenant un tableau et l'indice de la première case libre dans la pile.
5. Définir la fonction `new_stack` qui prend un entier en entrée et construit une pile de cette taille.
6. Ecrire une fonction `full_stack` qui teste si une pile est pleine.
7. Réécrire les primitives de base
8. Ajouter un mécanisme qui augmente la taille de la pile quand on ajoute un élément et qu'elle est pleine.

### 1.4 Calculette post-fixée

On désire implanter une mini-calculatrice post-fixée. On acceptera les 4 opérations arithmétiques habituelles.

Fonctionnement de la calculette :
Répéter :
- Lire une ligne.
- Si c'est un entier : empiler.
- Si c'est un opérateur : évaluer et afficher.
- Sinon : ignorer.

L'évaluation d'un opérateur supposera que les 2 opérandes se trouvent en sommet de pile.

1. Ecrire une fonction `reduce` de type `int stack -> char -> unit` où le second argument est un symbole d'opération et qui applique cette opération sur les deux premiers éléments de la pile pour les remplacer par le résultat obtenu.
2. Ecrire une fonction `cal` de type `unit -> unit` qui réalise la calculette.

### 1.5 Listes doublement chaînées

On se propose ici de définir des listes doublement chaînées, c'est-à-dire que chaque élément d'une liste connaît son successeur et son prédécesseur.
On définit les types suivants :

```
type 'a triplet = {elem : 'a;
                   mutable suiv : 'a ldc;
                   mutable prec : 'a ldc }
and  'a ldc = V | E of 'a triplet;;

type liste_dc = {tete:ldc; queue:ldc};;
```

1. Ecrire une fonction `ajoute` qui prend un élément et une `ldc` comme argument et retourne une `ldc` avec l'élément passé en tête.
2. Ecrire une fonction d'affichage d'une liste doublement chaînée.
3. Modifier cette fonction pour pouvoir afficher une liste doublement chaînée en partant de n'importe quel élément ce celle-ci.
4. Ecrire une fonction d'insertion d'un élément liste doublement chaînée d'entiers en respectant l'ordre croissant.

### 1.6 Simulation d'automates cellulaires

Le monde des cellules est représenté par un damier de n sur m cotés. Ce monde est un tore (chambre à air) : le voisin de droite d'un bord est la première case du bord opposé.
Il existe une population $G_0$. Le calcul de la génération $G_{n+1}$ se fait à partir de la génération $G_n$, d'une règle de création et d'une règle de destruction des cellules en fonction du voisinage.

**règle de destruction** : si une cellule a plus de 3 voisins ou moins de 2 voisins elle disparaît.

**règle de création** : si une case vide possède 3 voisins alors une cellule naît sur cette case.

```
     Gn                        Gn+1
-------------             -------------
| | | | | | |             | | | | | | |
-------------             -------------
| | | | | | |             | | |*| | | |
-------------     ==>     -------------
| |*|*|*| | |             | | |*| | | |
-------------             -------------
| | | | | | |             | | |*| | | |
-------------             -------------
```

1. Choisir une structure de données pour représenter un monde.
2. Ecrire une fonction `affiche_monde` d'affichage d'un monde.
3. Ecrire une fonction `nv_voisins` prenant un monde `m`, les coordonnées d'une case et qui retourne le nombre de voisins de cette case.

4. Ecrire une fonction `nouvelle_gen` qui prend un monde et calcule un nouveau monde contenant la nouvelle génération.
5. Ecrire une fonction `saisie_monde` qui demande au clavier la taille du monde et les coordonnées des cases remplies puis retourne ce monde.
6. Ecrire la fonction principale qui demande la saisie du monde, puis entre dans une boucle qui affiche la génération $n$, demande au clavier s'il faut continuer le calcul, si oui la calcule, si non sort de la fonction.

## 1.7 Graphes

On représente les graphes par des matrices d'adjacence.
Le graphe dont les arcs sont `[(0,1) ; (0,2) ; (1,2) ; (2,3) ; (3,0)]` est représenté par la matrice booléenne

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 |

On pose `type graphe =  bool array array`

1. Définir la fonction `graph_of` de type `(int * int) list -> graphe` qui transforme un graphe donné sous forme de liste de couples en son équivalent en matrice d'adjacence.
2. Ecrire une fonction `adjs_of` de type `int -> bool array array -> int list` telle que `adjs_of x m` donne la liste des sommets adjacents à `x`.
3. Ecrire une fonction qui calcule la clôture transitive du graphe.
4. Voici une version itérative du parcours en profondeur : on se donne une pile `p`, on commence le parcours au sommet `s0` et on remplit une liste `l` qui contiendra le parcours,

   `l` ← `[]` ; Empiler `s0` sur `p` ;
   Tant que `p` n'est pas vide faire
   Soit `s` le sommet de `p` ;
   Si il existe un sommet `x` adjacent à `s` et non encore visité alors Empiler `x`
   sinon Rajouter `s` à `l` ; Dépiler ;
   Fin-tant-que
   Imprimer `l`

   Ecrire une fonction `ipp` qui implante cet algorithme.
5. Le parcours en largeur est construit sur le même schéma, mais il utilise une file d'attente à la place de la pile. Ecrire la fonction `ipl` qui implante le parcours en largeur.

## 1.8 Valeurs potentiellement infinies

On cherche à décrire des ensembles potentiellement infinis.
Pour cela on utilisera le type paramétré suivant :

```
type 'a ensemble = {mutable cour:'a; suiv: 'a -> 'a;
                    mutable  dejavu : 'a list}
```

– `cour` : contient la dernière valeur calculée
– `suiv` : fonction de calcul de l'élément suivant
– `dejavu` : liste des valeurs déjà rencontrées

1. Ecrire la valeur de type `int ensemble` correspondant à l'ensemble des entiers pairs.
2. Ecrire une fonction polymorphe `suivant` de type `'a ensemble -> 'a` qui calcule une nouvelle valeur de cet ensemble en mettant à jour les champs `cour` et `dejavu`.
3. Ecrire une fonction `n_pair` qui retourne les $n$ premiers entiers pairs.
4. Déclarer la valeur `ens_prem` de type `int ensemble` correspondant à l'ensemble des nombres premiers. Pour tenir compte du champs `dejavu` il sera nécessaire d'effectuer une déclaration récursive.
5. Ecrire une fonction `interval_prem` qui prend deux bornes entières et retourne la liste des nombres premiers compris entre ces 2 bornes. On utilisera dans cette fonction la déclaration précédente.

## 1.9 Matrice des distances

**Le monde** Le monde est un rectangle composé de cases. On représente les cases par des caractères. Chaque case est soit vide ('.'), soit un mur ('#'), soit de l'eau ('~'). Les murs sont infranchissables. Si un personnage passe sur uune case eau, il se noie. Les bords du monde sont des murs. Les coordonnées du monde sont des paires d'entiers, allant de (1,1) à (largeur,hauteur), où (1,1) est le coint sud-ouest du monde.

| | | |
|---|---|---|
| .....<br>.....<br>##.~~~~<br>...~~~~<br>..... | .....<br>.....<br>.######.##<br>.....#...<br>.....##..<br>..... | ~~~~~~~~~~~~~~~<br>~.....~<br>~.....~<br>~..~~~...~~~..~<br>~..~.....~..~<br>~..~.....~..~<br>~~~~~~~~~~~~~~~ |
| carte 1 | carte 2 | carte 3 (ile) |

Fig. 1 – Exemples de cartes

On cherche à construire une matrice de distances entre une origine et une destination. On utilisera la distance Manhattan définie de la manière suivante :
– chaque case a 4 voisines dans les directions O, E, N et S
– chaque voisine d'une case est à une distance de 1 de celle-ci
On définit le type `mdist` suivant :

```
type mdist = int array array.
```

on cherche à écrire une fonction prenant un monde, une origine et une destination et construit la matrice des distances en commençant par la destination. Pour se rappeler des cases à traiter, il est conseillé de les conserver dans une structure linéaire comme une file d'attente.

1. Ecrire une fonction `distVoisins` qui prend un monde, une matrice des distances, une structure linéaire contenant des coordonnées de cases et les coordonnées d'une case qui possède déjà une distance à la destination qui pour chaque voisin effectue les actions suivantes :
   - si ce n'est pas une case libre : ne rien faire
   - sinon, si la distance de cette case à la destination est déjà connue, ne rien faire
   - sinon lui affecter la distance de la case + 1 et ajouter cette case dans la structure linéaire passée en argument.
2. En utilisant la fonction précédente, écrire une fonction `calcDist` qui prend un monde et une destination puis construit la matrice des distances par rapport à la destination.