Université Bordeaux 1 Licence S&TS Mention Informatique S5 2012/2013

UE INF5011 Programmation 3

Programmation Fonctionnelle et Symbolique

Projet

À remettre le vendredi 14 Décembre 2012

L'**énumeration** d'un ensemble d'objets discrets est incontournable quand l'ensemble est infini ou simplement trop grand pour être représenté *in extenso*.

1 Le type de donnée abstrait : `enumerateur`

D'un point de vue mathématique, un *énumerateur* est une machine dont les états sont les éléments d'une suite

$$(E_n)_{n\in\mathbb{N}}$$

d'éléments $e_0, e_1, \ldots$ d'un type donné. Un énumerateur peut être fini

$$(E_n)_{n\in[0,N]}, N \in \mathbb{N}$$

ou infini

$$(E_n)_{n\in\mathbb{N}}.$$

L'opération principale applicable à un énumérateur E est

$$\texttt{call-enumerator}(E)$$

qui retourne deux valeurs :
– $(e, true)$ s'il y avait un prochain élément e dans la suite,
– $(false, false)$ sinon.
Cette opération est destructive puisqu'elle change l'état interne de l'énumérateur.
Autrement dit, le premier appel retourne $(e_0, true)$, le second appel $(e_1, true)$ et ainsi de suite. Si la suite est de longueur n, le $n$ième appel retourne $(e_{n-1}, true)$ et tous les appels suivants $(false, false)$.
Un énumérateur E peut être réinitialisé par un appel à

$$\texttt{init-enumerator}(E)$$

Il recommence alors à énumérer à partir de e_0. Par commodité, `init-enumerator` retournera l'énumérateur. Cette opération est **destructive** puisqu'elle change l'état interne de l'énumérateur.

```
(defclass abstract-enumerator () ())

(defgeneric init-enumerator (enumerator)
  (:documentation
   "reinitializes and returns ENUMERATOR"))

(defgeneric copy-enumerator (enumerator)
  (:documentation
   "returns a reinitialized
    copy of ENUMERATOR"))

(defgeneric next-element-p (enumerator)
  (:documentation
   "returns NIL if there is no next
    element, a non NIL value otherwise"))

(defgeneric next-element (enumerator)
  (:documentation
   "returns the next element,
    moves to the following one"))

(defgeneric call-enumerator (enumerator)
  (:documentation
   "if there is a next element e,
    returns e and T
    and moves to the next element;
    otherwise returns NIL and NIL"))
```

FIGURE 1 – API for the `enumerator` abstract data type

L'opération

`copy-enumerator`(E)

retourne une copie réinitialisée d'un énumérateur E.
L'opération `call-enumerator` est implémentée pour tous les énumérateurs en utilisant deux opérations de plus bas niveau `next-element-p (enumerator)` et `next-element (enumerator)`. Ceci est présenté Figure 2.
L'opération `init-enumerator` est implémentée pour tous les énumérateurs comme montré Figure 3. Mais elle est destinée à être complétée par des méthodes secondaires (`:after` pour la plupart) selon la définition particulière d'un énumérateur concret.
Les opérations dont le nom est de la forme

`make-*-enumerator*`$(\ldots)$

```
(defmethod call-enumerator ((e abstract-enumerator))
  (if (next-element-p e)
      (values (next-element e) t)
      (values nil nil)))
```

FIGURE 2 – Implémentation of `call-enumerator`

```
(defmethod init-enumerator ((e abstract-enumerator))
  e)
```

FIGURE 3 – Implementation of `init-enumerator`

permettent de créer des énumérateurs concrets dépendant ou pas d'autres énumérateurs. L'API est donnée Figure 1.

2 Énumerateurs Simples

2.1 Éléments d'une liste

Les éléments de la suite sont les éléments d'une liste. L'ordre est l'ordre des éléments de la liste. Un exemple d'utilisation d'un tel énumerateur est montré Figure 4.

2.2 Suite définie par induction

Souvent une suite $(s_n)_{n\in\mathbb{N}}$ est définie de manière inductive avec une valeur initiale v et une fonction f qui calcule l'élément suivant à partir du précédent.

$$\left\{ \begin{array}{lcll} s_0 & = & v & \\ s_{n+1} & = & f(s_n) & \text{if } n > 0 \end{array} \right.$$

Example 2.1 *Par exemple, avec la valeur initiale $v = 1$ et*

$$\left\{ \begin{array}{ll} f: & \mathbb{N} \to \mathbb{N} \\ & x \mapsto 2 + x \end{array} \right.$$

on obtient la suite des entiers naturels impairs.

$$1, 3, 5, \ldots$$

On peut ajouter à un énumérateur inductif une fonction g à appliquer après f.

Example 2.2 *Avec un tel énumérateur, on peut facilement générer un suite infinie de la forme*

$$0, 1, \ldots n-1, 0, 1, \ldots$$

comme montré Figure 5 avec $n = 2$.

```
ENUM> (setf *e* (make-list-enumerator '(1 2 3)))
#<LIST-ENUMERATOR {1004711A71}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (init-enumerator *e*)
#<LIST-ENUMERATOR {1005AF15F1}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (call-enumerator *e*)
3
T
ENUM> (call-enumerator *e*)
NIL
NIL
ENUM> (call-enumerator *e*)
NIL
NIL
```

FIGURE 4 – Exemple d'énumérateur des éléments d'une liste

```
  ENUM> (setf *e*
              (make-mod-inductive-enumerator 0 #'1+ (lambda (x) (mod x 2))))
#<MOD-INDUCTIVE-ENUMERATOR {1002A7E1A1}>
ENUM> (call-enumerator *e*)
0
ENUM> (call-enumerator *e*)
1
ENUM> (call-enumerator *e*)
0
ENUM> (call-enumerator *e*)
1
```

FIGURE 5 – Exemple of a modulo inductive enumerator

3 Combinaisons d'énumérateurs

Nous notons $\mathbb{E}$ l'ensemble des énumérateurs. Les énumérateurs peuvent se combiner pour donner de nouveaux énumérateurs.
Par exemple, si on a deux énumérateurs E_1 et E_2, énumérant respectivement $e_0^1, e_1^1 \ldots$ et $e_0^2, e_1^2 \ldots$, on peut vouloir faire de la concaténation des deux suites, le produit cartésien ou le produit parallèle, etc.
Dans cette section, on va définir des opérateurs s'appliquant à un ou plusieurs énumérateurs pour donner un nouvel énumérateur.

Definition 1 *Un* opérateur *est une application*

$$\left\{ \begin{array}{rcl} \mathbb{E}^n & \rightarrow & \mathbb{E} \\ E_1, \ldots E_n & \mapsto & E \end{array} \right.$$

Certains opérateurs seront unaires (de $\mathbb{E}$ dans $\mathbb{E}$), d'autres binaires (de $\mathbb{E}^2$ dans $\mathbb{E}$) et d'autres encore d'arité variable (de $\bigcup_{k\in\mathbb{N}} \mathbb{E}^i$ dans $\mathbb{E}$). La figure 6 montre des opérateurs classiques.

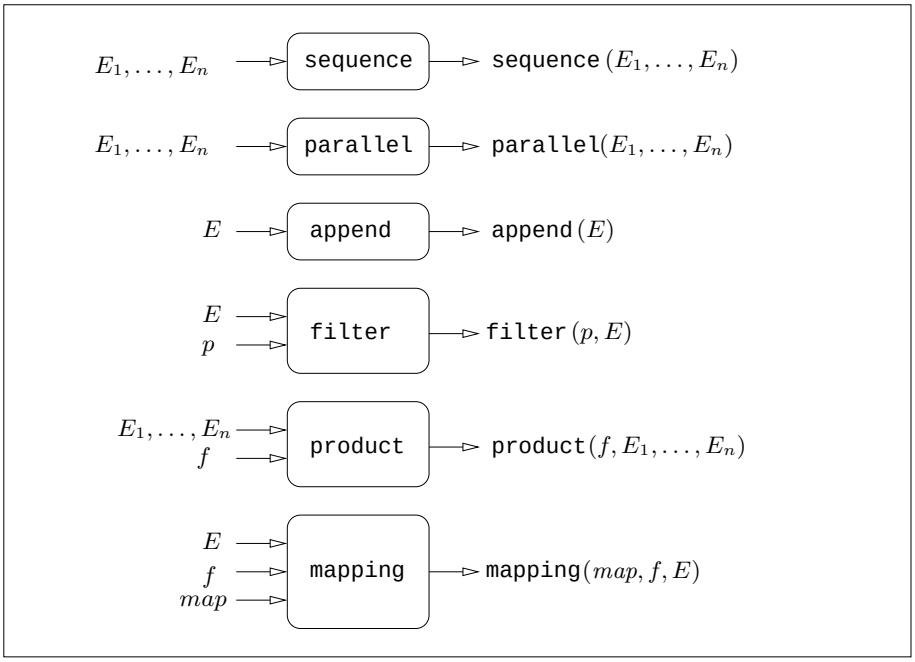

FIGURE 6 – Combinaisons d'énumérateurs

3.1 Opérateurs d'application

Supposons que E énumère $e_0, e_1, \ldots$ et que f soit une fonction telle que tout e_i est dans son domaine de définition. On voudrait un énumérateur $E' = \texttt{funcall}(f, E)$ qui énumère

$$f(e_0), f(e_1), \ldots$$

```
(defun make-funcall-enumerator (fun e)
  (make-mapping-enumerator fun #'funcall e))

(defun make-apply-enumerator (fun e)
  (make-mapping-enumerator fun #'apply e))
```

FIGURE 7 – Énumérateurs `funcall` et `apply`

```
(make-funcall-enumerator
 (lambda (x) (* x x))
 (make-inductive-enumerator 0 #'1+))
```

FIGURE 8 – Carré des entiers naturels

f étant un paramètre, $f(e_i)$ pourra être calculé grâce à l'appel

$$(\texttt{funcall}\ f\ e_i)$$

De même, si E énumère $l^0, l^1, \ldots$ tels que chaque l^i est une liste $(e_1^i, e_2^i, \ldots, e_{k_i}^i)$ et f est une fonction acceptant un nombre quelconque d'arguments, on voudrait un énumerateur $E' = \texttt{apply}(f, E)$ énumérant

$$f(e_1^0, e_2^0, \ldots, e_{k0}^0),$$
$$f(e_1^1, e_2^1, \ldots, e_{k_1}^1), \ldots$$

f étant un paramètre et chaque $l^i = (e_1^i, e_2^i, \ldots, e_{k_i}^i)$ une liste, $f(e_1^i, e_2^i, \ldots, e_{k_i}^i)$ peut être calculé par l'appel (`apply` f l^i).
Ces deux types d'énumérateurs peuvent être obtenus grâce à un opérateur unique dit de *mapping* (ou d'application) paramétré par une *fonction de mapping* :

$$\texttt{mapping}(map, f, E)$$

La fonction de *mapping* sera le plus souvent `funcall` ou `apply`.
Les deux types d'énumérateurs décrits au début de la section sont un cas particulier de l'opérateur de `mapping`, l'un utilisant `funcall` comme fonction de mapping et l'autre `apply`.

Example 3.1 *Si on veut un énumérateur des carrés des entiers naturels*

$$0, 1, 4, 9, 16, \ldots$$

on peut écrire l'expression de la Figure 8.

La fonction Lisp `reduce` peut aussi être utilisée comme fonction de mapping.

3.2 Séquence

Étant donnée une suite de n énumerateurs $E_1, E_2, \ldots, E_n$, énumérant chacun

$$e_0^i, e_1^i, \ldots, e_{k_i}^i,$$

on voudrait un énumerateur

$$E = \texttt{sequence}(E_1, E_2, \ldots, E_n)$$

qui énumère d'abord tous les éléments de E_1, puis tous les éléments de E_2 et ainsi de suite, *i.e.*

$$\begin{array}{l} e_0^1, e_1^1, \ldots, e_{k_1}^1, \\ e_0^2, e_1^2, \ldots, e_{k_2}^2, \\ \ldots \\ e_0^n, e_1^n, \ldots, e_{k_n}^n, \end{array}$$

L'énumérateur $\texttt{sequence}(E_1, E_2, \ldots, E_n)$ est fini si et seulement si chaque E_i est fini.

Example 3.2 *Par exemple, avec $n = 2$, E_1 énumérant a, b, c et E_2 énumérant $1, 2, \ldots$ alors $\texttt{sequence}(E_1, E_2)$ énumère $a, b, c, 1, 2, \ldots$*

3.3 Parallélisation

Étant donnée une suite de n énumerateurs $E_1, E_2, \ldots, E_n$ énumérant chacun $e_0^i, e_1^i, \ldots, e_{k_i}^i$, on voudrait un énumérateur

$$E = \texttt{parallel}(E_1, \ldots, E_n)$$

qui énumère les n-uples

$$\begin{array}{l} (e_0^1, e_0^2, \ldots, e_0^n) \\ (e_1^1, e_1^2, \ldots, e_1^n) \\ \ldots \\ (e_k^1, e_k^2, \ldots, e_k^n) \end{array}$$

où $k = min(k_1, \ldots, k_n)$. L'énumerateur

$$\texttt{parallel}(E_1, \ldots, E_n)$$

est infini si et seulement si tous les E_i sont infinis.

4 Énumérateurs nécessitant de la mémorisation

Supposons que nous ayons une suite de n énumérateurs $E_1, E_2, \ldots, E_n$, chaque énumérateur E_i énumérant

$$e_0^i, e_1^i, \ldots$$

On voudrait un énumerateur $E = X(E_1, \ldots, E_n)$ énumérant les n-uples du produit cartésien des listes des valeurs énumérées par chacun des E_i *i.e.*

$$\begin{array}{llll}
(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_0^n) & (e_0^1, e_0^2, \ldots, e_0^{n-1}, e_1^n) & \ldots & (e_0^1, e_0^2, \ldots, e_0^{n-1}, e_{k_n}^n) \\
(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_0^n) & (e_0^1, e_0^2, \ldots, e_1^{n-1}, e_1^n) & \ldots & (e_0^1, e_0^2, \ldots, e_1^{n-1}, e_{k_n}^n) \\
\ldots & \ldots & \ldots & \\
(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_0^n) & (e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_1^n) & \ldots & (e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n)
\end{array}$$

$X(E_1, \ldots, E_n)$ est fini si et seulement si tous les E_is sont finis. Si E_i est infini alors seulement un élément de chacun des énumérateurs précédents $E_1, \ldots, E_{i-1}$ sera énuméré.

Example 4.1 *Par exemple, avec* $n = 2$*,* E_1 *énumérant* $1, 2, \ldots$ *et* E_2 *énumérant* a, b, c *alors* $X(E_1, E)$ *énumère* $(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), \ldots$

Il n'est pas plus difficile et plus général de créer un énumérateur

$$E = \texttt{product}(f, E_1, \ldots, E_n)$$

énumérant les valeurs de f appliquée aux n-uples du produit cartésien *i.e.*

$$\begin{array}{llll}
f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_0^n) & f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_1^n) & \ldots & f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_{k_n}^n) \\
f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_0^n) & f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_1^n) & \ldots & f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_{k_n}^n) \\
\ldots & \ldots & \ldots & \\
f(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_0^n) & f(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_1^n) & \ldots & f(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n)
\end{array}$$

Alors $X(E_1, \ldots, E_n)$ est un cas particulier de $\texttt{product}(f, E_1, \ldots, E_n)$ en prenant $f = \texttt{list}$. Pour énumérer les valeurs de E_i, on doit se souvenir des valeurs de $E_1, \ldots, E_{i-1}$. Pour cela, on pourra créer un type d'énumérateur (`memo-enumerator` permettant de mémoriser la dernière valeur énumérée. Cette valeur sera disponible tant qu'elle n'est pas écrasée par un appel à `next-element`.

4.1 Énumérateur à mémoire

Étant donné un énumérateur E qui énumère $e_0, e_1, \ldots$, on souhaite un énumérateur $E' = \texttt{M}(E)$ qui énumère les mêmes valeurs mais avec possibilité de répéter autant de fois que nécessaire chaque e^i. Cet énumérateur aura donc un créneau permettant de mémoriser la dernière valeur énumérée. Ce créneau sera initialement non affecté. Si le créneau n'est pas affecté, l'appel à `next-element` positionnera le créneau à la valeur suivante de E. Les appels suivants retourneront toujours cette même valeur jusqu'à ce que le créneau soit désaffecté (opération `unset-memo-object (memo-enumerator)`). Un exemple est donné Figure 9.

```
ENUM> (defparameter *e* (make-list-enumerator '(1 2 3)))
*E*
ENUM> (defparameter *m* (make-memo-enumerator *e*))
*M*
ENUM> (call-enumerator *m*)
1
T
ENUM> (call-enumerator *m*)
1
T
ENUM> (call-enumerator *m*)
1
T
ENUM> (unset-memo-object *m*)
#<MEMO-ENUMERATOR {1005A345B1}>
ENUM> (call-enumerator *m*)
2
T
ENUM> (call-enumerator *m*)
2
T
ENUM> (unset-memo-object *m*)
#<MEMO-ENUMERATOR {1005A345B1}>
ENUM> (call-enumerator *m*)
3
T
ENUM>
```

FIGURE 9 – Exemple d'énumérateur à mémoire

```
ENUM> (setf *e* (make-list-enumerator '(1 2) t))
#<LIST-ENUMERATOR {1004742B71}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (call-enumerator *e*)
1
T
```

FIGURE 10 – Exemple d'énumérateur de liste circulaire

4.2 Concaténation

Parfois, on a un énumerateur énumérant des listes $l^0, l^1, \ldots$ avec chaque $l^i = (l^i_0, l^i_1, \ldots, l^i_{k_i})$ mais on voudrait un énumérateur des éléments de la concaténation des listes :

$$l^0_0, l^0_1, \ldots, l^0_{k_0}, l^1_0, l^1_1, \ldots, l^1_{k_1}, \ldots$$
$$l^i_0, l^i_1, \ldots, l^i_{k_i}, \ldots$$

mais en évitant naturellement de construire in extenso les listes et leur concaténation.

4.3 Filtrage

Étant donné un énumérateur E et un prédicat p, il est facile d'implementer un énumérateur

$$\texttt{filter}(p, E)$$

qui énumère la sous-suite d'éléments de E qui satisfait p comme `remove-if-not` sur une `sequence Lisp`.

Example 4.2 *Soit E un énumérateur de $\mathbb{N}$. L'énumérateur $\texttt{filter}_{primep}$, avec `primep` un prédicat testant la primalité, énumère l'ensemble des nombres premiers $2, 3, 5 \ldots,$.*

5 Travail demandé

1. Implémenter `list-enumerator`.
2. Fournir une opération

   `make-list-enumerator(l &optional circ)`

   Quand le paramètre optionel `circ` est positionné à `T`, l'énumérateur produit est infini : il énumère circulairement les objets de `l` comme indiqué Figure 10.

3. Implémenter les énumérateurs inductifs.
4. Implémenter les énumérateurs parallèles.
5. Implémenter le filtrage.
6. Implémenter la concaténation.
7. Facultatif :
   (a) Implémenter les énumérateurs à mémoire
   (b) Implémenter le produit cartésien.

6 Modalités

1. Le travail doit être réalisé en trinômes.
2. L'ensemble des fichiers associés au projet doit être gérés avec un outil de gestion de révision (svn, git, ...) : directement dans répertoire d'accueil ou sur le serveur `Savanne` du `CREMI`

   `https://services.emi.u-bordeaux1.fr/projet/savane/`
3. On demande un rapport de 5 pages maximum présentant les choix effectués, les résultats et les tests.
4. Chaque enseignant de TD proposera sa solution pour la remise du projet (archive, mail, soutenance,...).