Auteur : R. Namyst
Enseignants : L. Bouge, A. Bonhomme
Email : {Luc.Bouge,Alice.Bonhomme}@ens-lyon.fr

le 02/10/00

TD Programmation 1 : Listes et itérateurs

1 Utilisation de caml

Le lancement de l'interpréteur CAML s'effectue au moyen de la commande `ocaml` (pour *Objective* CAML). Le manuel de référence du langage est disponible "en ligne" à l'URL :

`http://caml.inria.fr/ocaml/htmlman/index.html`

Il est conseillé d'ouvrir un éditeur de texte (par exemple `emacs`) depuis lequel les commandes CAML seront progressivement copiées et collées dans la fenêtre de l'interpréteur.

2 Fonctions simples sur les listes

2.1 Le type abstrait `ilist`

Définissez le type `ilist` suivant :

```
type ilist = Empty | Cons of int * ilist ;;
```

Munissez le type `ilist` de quelques opérations élémentaires de manière à pouvoir le manipuler comme un type abstrait :

`empty : unit -> ilist`

Retourne une liste vide.

`cons : int -> ilist -> ilist`

Construit une liste à partir d'une tête (un entier) et d'une queue (une liste).

`car : ilist -> int`

Retourne l'élément en tête de liste.

`cdr : ilist -> ilist`

Retourne la liste amputée du premier élément.

`is_empty : ilist -> bool`

Indique si la liste est vide.

2.2 Fonctions simples

En utilisant les opérations élémentaires précédentes, définissez les fonctions suivantes :

`length : ilist -> int`
Retourne la longueur de la liste.
`equal : ilist -> ilist -> bool`
Indique si les listes sont semblables.
`reverse : ilist -> ilist`
Retourne la liste miroir.
`append : ilist -> ilist -> bool`
Retourne la concaténation des deux listes.

2.3 Récursivité terminale

Pour chacune des fonctions précédentes, donnez si possible une version à récursivité terminale.

3 Itérateurs sur les listes

On considère maintenant des listes CAML au sens général et l'on s'autorise donc le *filtrage* lors de l'écriture des fonctions suivantes.

3.1 Itérateurs proceduraux

Définissez les itérateurs suivants et testez-les à l'aide d'une fonction `f` se réduisant à l'affichage de son argument sur la sortie standard.

`dolist : ('a -> 'b) -> 'a list -> unit`
`dolist f [a1;...;an]` exécute l'équivalent de `f a1; f a2; ... f an; ()`.

`dolistif : ('a -> bool) -> ('a -> unit) -> 'a list -> unit`
`dolistif t f [a1;...;an]` évalue en séquence les expressions `... ; f ai ; ... ; ()` vérifiant `t ai`.

`forlist : int -> int -> ('a -> 'b) -> 'a list -> unit`
`forlist i j f [a1;...;an]` évalue en séquence les expressions `f ai ; ... ; f aj ; ()`. Si $i > j$, seule l'expression `()` est évaluée.

3.2 Itérateurs fonctionnels

Définissez les itérateurs suivants :

`map : ('a -> 'b) -> 'a list -> 'b list`
`map f [a1; ...;an]` construit la liste `[f a1; ...;f an]`.

`select : ('a -> bool) -> 'a list -> 'a list` `select t [a1;...;an]` construit la liste `[...;ai;...]` des éléments vérifiant `t ai`.

`reduce : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
`reduce f [a1;...;an] b` calcule `f a1 (f a2 ( ...(f an b)... ))`.

`reduce_bis : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`reduce_bis f a [b1;...;bn]` calcule `f (... f (f a b1) b2 ...) bn`.

`scan : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b list`
`scan f [a1;...;an] b` construit la liste `[f a1 (... (f an b)...) ;...;f an b]`.

Assurez-vous que la fonction `scan` n'effectue pas de calculs redondants. De manière générale, veillez également à minimiser le nombre d'arguments lors des appels récursifs.

Vérifiez l'ordre d'évaluation des éléments lors d'un appel à `map` à l'aide d'une fonction `f` affichant une trace d'exécution sur la sortie standard. Modifiez la fonction `map` pour inverser l'ordre d'évaluation.

3.2.1 Récursivité terminale

Pour chacune des fonctions précédentes, donnez si possible une version à récursivité terminale.

3.2.2 `reduce` et `reduce_bis` revisités

Réécrivez l'itérateur `reduce` à l'aide de l'itérateur `reduce_bis` et de la fonction `reverse`. Même question pour `reduce_bis` en utilisant `reduce` et `reverse`.

3.2.3 Une fonction à récursivité non-terminale ?

On se propose d'étudier une instance particulière du problème bien connu du "sac à dos". En l'occurence il s'agit, à partir d'un ensemble d'entiers positifs $l = \{a_1; a_2; ...; a_n\}$ (représentant les "poids" des objets) et d'un entier naturel b (la "contenance" du sac), d'extraire un sous-ensemble $s = \{...; a_i; ...\} \subseteq l$ dont la somme des éléments Σ_s est inférieure ou égale à b et vérifie : $\forall s' \subseteq l, \Sigma_{s'} \leq b \Rightarrow \Sigma_{s'} \leq \Sigma_s$.

Écrivez une fonction `meilleur : int list -> int -> int list` qui, à partir d'une liste l et un entier b, retourne la sous-liste vérifiant la propriété énoncée ci-dessus.

4 Itérateur générique

On se propose maintenant de définir un itérateur général `boucle` paramétré par une fonction "test d'arrêt", par une fonction à itérer et par l'argument initial de cette fonction. En voici la définition :

```
let rec boucle t f x = if (t x) then x else boucle t f (f x) ;;
```

Utilisez cet itérateur pour définir la fonction `fibonacci(n)`. Veillez à ne calculer qu'une seule fois chaque élément de la suite.