Algorithmique fonctionnelle - Listes

11 février 2013

Rappels

Listes

La liste est la structure de donnée au centre du paradigme fonctionnel.
Une liste est définie comme une structure de données **linéaire** et **dynamique** d'éléments **de même type**.
C'est une structure **récursive** qui peut être :

- Vide
- La juxtaposition d'un élément, la **tête** et d'une autre liste, le **reste**

Rappels

Listes

Algorithmiquement, les fonctions sur les listes, permettant de les manipuler sont :

- **cons**(e : Element, l Liste d'Elements) : Liste d'Element
- **estVide**(l Liste d'Elements) : booléen
- **tete**(l Liste d'Elements) : Element
- **reste**(l Liste d'Elements) : Liste d'Elements

Ocaml

Listes

Il est possible de construire une liste, comme vu en Turbo Pascal, au moyen d'un type récursif.

Liste d'entiers, type récursif

```
type liste = Maillon of int * liste
           | Nul;;
```

Ocaml

Listes

Une liste en Ocaml est une structure existante et manipulable comme définie en algorithmique.
Éléments du langage permettant de construire une liste :

Éléments du langage

[ ] : la constante liste vide
:: : le constructeur de liste

Ocaml

Exemple

```
let l = [ ];;
let l = [3; 7; 12];;
let l = 8 :: [3; 7; 12];;
let m = 7 :: 5 :: [ ];;
let l = 4 :: 2. ::[ ];;
let m = [(function x -> x **2.); (function y -> y - 3)];;
let l = [3, 7, 12];;
```

Fonctions de base

Tête et reste

Une liste est manipulable par les fonctions du module `List`.

- `List.hd` : retourne la tête de la liste
- `List.tl` : retourne le reste de la liste

Fonctions de base

Exemple

```
let l = [3; 8; 5] ; ;
List.hd l ; ;      (* 3 *)
List.tl l ; ;      (* [8; 5] *)
```

Fonctions de base

Pattern-matching

Le pattern-matching permet d'identifier la tête et le reste d'une liste.

Fonction qui verifie qu'une liste est uniquement remplie de zéros

```
let rec zero l =
    match l with
    [ ] -> true
    | 0 : :r -> zero r
    | _ : :r -> false ; ;
```

Fonctions de base

Fonction longueur d'une liste

```
let rec longueur l =
    match l with
    [ ] -> 0
    | _ : :r -> 1 + longueur r ; ;
```

Fonctions de base

Fonction longueur d'une liste

```
let rec amaigrit l =
    match l with
    a : : (b : : _ as r) -> if (a = b) then amaigrit r else a : :
amaigrit r
    | _ -> l ; ;
```

Fonctions de base

Concaténation

Opérateur de concaténation : `@`

Concaténation

```
let l = [3; 12; 7] ; ;
let m = [2; 8] ; ;
let n = l @ m ; ;
```

Fonctions avancées

Autres fonctions

- `List.length` : longueur de la liste
- `List.rev` : renverse la liste
- `List.mem` : vérifie qu'un élément appartient à la liste
- `List.nth` : retourne le i^e élément de la liste
- ...

Fonctions avancées

La fonction `List.map`

La fonction `List.map` permet d'appliquer une fonction **f** à chaque élément de la liste **f**.
Elle retourne une **nouvelle** liste dont chaque élément correspond à l'image de l'élément correspondant de **l** par la fonctionf **f**.

Exemple

```
let l = [3; 12; 7] ; ;
let f x = x * x ; ;
List.map f l ; ;
```

Pouvons-nous écrire une telle fonction **mamap** ?

Fonctions avancées

Exercice

Bien sûr ! !, Voici la fonction **mamap** :

```
let rec mamap f l =
match l with
[ ] -> [ ]
| t : :r -> (f t) : : mamap f r ; ;
```

Fonctions avancées

Les fonctions `fold_left` et `fold_right`

Les fonctions `fold` sont des fonctions d'aggrégat qui appliquent une fonction binaire à chacun des membres de la liste, de la manière suivante :

`fold_left f a` $[e1; e2; e3; ...; en]$ `= (f (f ... (f (f a` e_1`)` e_2`)` `...` e_{n-1}`)` e_n`)`

Exercice

fold_left (fun x y -> x + y) 2 [3; 7; 5]

let mystère l = List.fold_left (fun a b -> if b > a then b else a) (List.hd l) l

Fonctions avancées

Les fonctions `for_all` et `exists`

Les fonctions `for_all` et `exists` permettent de verifier un prédicat sur une ou toutes les valeurs de la liste

Exemple

let l = [3; 5; 7; −2] ; ;

```
List.for_all (fun x -> x mod 2 = 1) l;;
List.exists (fun x -> x <= 0) l;;
```

Fonctions avancées

Les fonctions `find` et `find_all`

Les fonctions `find` et `find_all` permettent d'appliquer un prédicat à tous les éléments de la liste et de retourner, celui ou ceux qui le vérifient.

Exemple

let l = [3; 5; 7; −2] ; ;

```
List.find (fun x -> x >= 0) l;;
List.find_all (fun x -> x >= 0) l;;
```

Compilation

Éléments

- Création d'un fichier à l'extension **.ml**
- Compilation : `ocamlc -o fichexe Fic.ml`
- Exécution : **./fichexe**

Compilation

Librairies

- Création d'un fichier librairie à l'extension **.ml**
- Compilation de ce fichier : `ocamlc -c Fic.ml` → Fic.cmo
- Compilation du fichier principal, avec les librairies : `ocamlc -o ficexe Fic.cmo Main.ml`

Librairies

Exemple

Permet de regrouper des fonctionnalités dans un même fichier et de les réutiliser :

```
(* fichier ListeTools.ml *)
  let rec mamap f l =
    match l with
    [ ] -> [ ]
    | t : : r -> (f t) : : mamap f r ; ;

  let rec listeEnChaine l =
    match l with
    [ ] -> "[ ]" ^"\n"
    | t : : r -> string_of_int(t) ^"-" ^listeEnChaine r ; ;
```

Librairies

Exemple

Permet de regrouper des fonctionnalités dans un même fichier et de les réutiliser :

```
(* fichier Main.ml *)
let l=[3;5;2;7;8] ;;
print_endline (ListeTools.listeEnChaine
                (ListeTools.mamap (fun x -> x * x) l)) ;;
```

Intéractions avec l'utilisateur

Fonctions

- `print_float`, `print_int`, `print_char`, `print_string`
- `print_newline`, `print_endline`
- `read_line`, `read_int`, `read_float`