

## Les données

## Programmation fonctionnelle

## Modules et compilation

David Teller

12/02/2007

Employons le type suivant pour représenter les formules ensemblistes:

```

type formule_ensembliste =
| Nom of string
| Union of formule_ensembliste * formule_ensembliste
| Intersection of Formule_ensembliste * Formule_ensembliste
| Complement of formule_ensembliste

```



## L'objectif

## Une solution (1 : les cas simples)

```

let rec descendre_complements = function
| Nom n      -> None (=On ne peut rien descendre.+)
| Complement (Intersection (a,b)) ->
  Some (Union (Complement a, Complement b))
| Complement (Union (a,b)) ->
  Some (Intersection (Complement a, Complement b))
| Complement a ->
  (
  match descendre_complements a with
  | None -> None
  | Some b -> Some (Complement b)
  )
(= ... =)

```



## Une solution (2 : l'union)

```
| Union (a,b) ->
  (
    match descendre_complements a with
    | Some c -> Some (Union (c,b))
    | None   ->
      (
        match descendre_complements b with
        | Some d -> (Some (Union (a,d)))
        | None   -> None
      )
  )
```

## Une solution (3 : l'intersection)

```
| Intersection (a,b) ->
  (
    match descendre_complements a with
    | Some c -> Some (Intersection (c,b))
    | None   ->
      (
        match descendre_complements b with
        | Some d -> (Some (Intersection (a,d)))
        | None   -> None
      )
  )
```

## Note

Ceci est un exemple typique de réécriture d'arbre. Ce genre de techniques est utilisé pour

- Maple** toutes les opérations formelles de dérivation, intégration et simplification d'expressions ressemblent à cela
- Java** le logiciel qui exécute les programmes Java ressemble à cela (mais écrit en C)
- Python** le logiciel qui exécute les programmes Python ressemble à cela (écrit en Haskell)
- Mozilla** la définition de la prochaine version de JavaScript ressemble à cela (écrit en SML).

## Note

Ceci est un exemple typique de réécriture d'arbre. Ce genre de techniques est utilisé pour

- Maple** toutes les opérations formelles de dérivation, intégration et simplification d'expressions ressemblent à cela
- Java** le logiciel qui exécute les programmes Java ressemble à cela (mais écrit en C)
- Python** le logiciel qui exécute les programmes Python ressemble à cela (écrit en Haskell)
- Mozilla** la définition de la prochaine version de JavaScript ressemble à cela (écrit en SML).

La réécriture d'arbre est quelque chose d'horrible en Java, pire encore en C/C++ ou en Python. C'est le domaine typique dans lequel excelle la programmation fonctionnelle (Haskell, OCaml ou SML).

## Note

Ceci est un exemple typique de réécriture d'arbre. Ce genre de techniques est utilisé pour

- Maple** toutes les opérations formelles de dérivation, intégration et simplification d'expressions ressemblent à cela
- Java** le logiciel qui exécute les programmes Java ressemble à cela (mais écrit en C)
- Python** le logiciel qui exécute les programmes Python ressemble à cela (écrit en Haskell)
- Mozilla** la définition de la prochaine version de JavaScript ressemble à cela (écrit en SML).

La réécriture d'arbre est quelque chose d'horrible en Java, pire encore en C/C++ ou en Python. C'est le domaine typique dans lequel excelle la programmation fonctionnelle (Haskell, OCaml ou SML).

La majorité des programmeurs ne sont pas capables de faire ce genre de choses correctement.



## Note

Ceci est un exemple typique de réécriture d'arbre. Ce genre de techniques est utilisé pour

- Maple** toutes les opérations formelles de dérivation, intégration et simplification d'expressions ressemblent à cela
- Java** le logiciel qui exécute les programmes Java ressemble à cela (mais écrit en C)
- Python** le logiciel qui exécute les programmes Python ressemble à cela (écrit en Haskell)
- Mozilla** la définition de la prochaine version de JavaScript ressemble à cela (écrit en SML).

La réécriture d'arbre est quelque chose d'horrible en Java, pire encore en C/C++ ou en Python. C'est le domaine typique dans lequel excelle la programmation fonctionnelle (Haskell, OCaml ou SML).

La majorité des programmeurs ne sont pas capables de faire ce genre de choses correctement. Ce qui ne vous empêchera probablement pas d'avoir quelques questions de ce genre à l'examen.



## Insertion

**Q** Écrire une fonction qui insère un élément à sa place dans une liste ordonnée, c'est-à-dire de manière à obtenir une nouvelle liste ordonnée.

### Exemple

```
# insere 'f' ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s']
- : char list = ['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p';'q';'r';'s']
# insere 7 [1;2;4;8;16;32]
- : int list = [1;2;4;7;8;16;32]
```



## Une insertion

```
let rec insere a "dans:t" = match t with
| [] -> [a]
| h::_ when a <= h -> a::t
| h::T -> h::(insere a "dans:t")
```



## Tri par insertion

Q À partir de la fonction précédente, écrire une fonction qui trie une liste.

### Exemple

```
* trie_liste [0;7;0;3;4;2]::
- : int list = [2;4;0;7;0;3]
```

## Un tri par insertion

```
let rec trie_liste l = match l with
| [] -> []
| h::t -> insere h "dans:(trie_liste t)
```

## Variante sur le thème

On peut mettre à contribution le système de types de OCaml pour s'assurer qu'on ne se mélange pas accidentellement listes triées et listes non triées.

## Variante sur le thème

On peut mettre à contribution le système de types de OCaml pour s'assurer qu'on ne se mélange pas accidentellement listes triées et listes non triées.

```
type 'a liste_triee = ListeTriee of 'a list

# let insere2 e "dans:(ListeTriee l)" =
  let rec aux l = match l with
  | [] -> [e]
  | h::_ when e <= h -> e::l
  | h::t -> h::(aux t)
  in ListeTriee (aux l);;
insere2 : 'a -> dans: 'a liste_triee -> 'a liste_triee = <fun >

# let rec trie2 l = match l with
| [] -> ListeTriee []
| h::t -> insere2 h "dans:(trie2 t);;
trie2 : 'a list -> 'a liste_triee = <fun >
```

## Variante sur le thème

On peut mettre à contribution le système de types de OCaml pour s'assurer qu'on ne se mélange pas accidentellement listes triées et listes non triées.

```
type 'a liste_triee = ListeTrie of 'a list

# let insere2 a ~dans:(ListeTrie t) =
  let rec aux l = match l with
  | [] -> [a]
  | h::_ when a <= h -> a::l
  | h::_ -> h::(aux t)
  in ListeTrie (aux t);
insere2 : 'a -> dans:'a liste_triee -> 'a liste_triee = <fst>

# let rec trie2 l = match l with
  | [] -> ListeTrie []
  | h::t -> insere2 h ~dans:(trie2 t);
trie2 : 'a list -> 'a liste_triee = <fst>
```

N'hésitez pas à recourir à ce genre de techniques pour vous construire des types de données spécifiques à une circonstance.

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

A Vous n'avez plus moyen de mettre une liste non triée par accident là où on attend une liste triée.

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

A Vous n'avez plus moyen de mettre une liste non triée par accident là où on attend une liste triée.

Q Enfin presque... quelle est la limitation ?

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

A Vous n'avez plus moyen de mettre une liste non triée par accident là où on attend une liste triée.

Q Enfin presque...quelle est la limitation ?

A Vous (ou quelqu'un qui n'aurait pas lu la documentation) pouvez prendre n'importe quelle liste `l` et en faire une `ListeTriee l` – même si elle n'est pas effectivement triée.

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

A Vous n'avez plus moyen de mettre une liste non triée par accident là où on attend une liste triée.

Q Enfin presque...quelle est la limitation ?

A Vous (ou quelqu'un qui n'aurait pas lu la documentation) pouvez prendre n'importe quelle liste `l` et en faire une `ListeTriee l` – même si elle n'est pas effectivement triée.

Q Comment empêcher cela ?

## Jusqu'où ?

Q Quel est l'intérêt de la manipulation ?

A Vous n'avez plus moyen de mettre une liste non triée par accident là où on attend une liste triée.

Q Enfin presque...quelle est la limitation ?

A Vous (ou quelqu'un qui n'aurait pas lu la documentation) pouvez prendre n'importe quelle liste `l` et en faire une `ListeTriee l` – même si elle n'est pas effectivement triée.

Q Comment empêcher cela ?

A Il faut que la seule méthode pour créer une `ListeTriee` vérifie que la liste est bien triée.

Réécriture

Tri par insertion  
Corrections

Modules  
Compilation

## Modules

Un *module* est un ensemble de types et de valeurs qu'on a mis à part dans le programme.

## Modules

Un *module* est un ensemble de types et de valeurs qu'on a mis à part dans le programme.

On sépare un projet en modules pour

- ▶ assurer un minimum la propreté du code
- ▶ permettre à plusieurs programmeurs de travailler en même temps sur des fonctionnalités différentes
- ▶ permettre de remplacer une partie du programme par une autre sans avoir à modifier le reste du programme.

## Définition de modules

Un module consiste en

**interface** la liste des types et, pour chaque nom de valeur, son type  
– un fichier `.mli`

**implantation** le détail des types et des valeurs – un fichier `.ml`.

## Définition de modules

Un module consiste en

**interface** la liste des types et, pour chaque nom de valeur, son type  
– un fichier `.mli`

**implantation** le détail des types et des valeurs – un fichier `.ml`.

Si vous ne fournissez pas le fichier `.mli`, OCaml l'inventera pour vous. Mais ce ne sera pas forcément ce que vous voulez : OCaml ne prendra jamais l'initiative de cacher le fait que `t` est un synonyme de `list`.

## Interface

```
(= Fichier liste_triee.mli=)

[==
Le type d'une liste triée.
=]
type 'a t

[==
Convertit une liste déjà triée en Liste_triee.t .

Cette fonction ne trie pas la liste. Elle se contente d'encapsuler
une liste déjà triée.
=]
val list_of_t : 'a t -> 'a list

[==
Convertit une liste en liste triée
=]
val t_of_list : 'a list -> 'a t
```

⏪ ⏩ ⏴ ⏵ 🔍

## Implantation

```
(= Fichier Liste_triee.ml =)
[== Une Liste_triee est, en tout et pour tout, une liste. ==]
type 'a t = 'a list

[==
Vérifie qu'une liste est triée
=]
let rec verifie_triee = function
| []      -> true
| h::[]   -> true
| h::i::t when h <= i -> verifie_triee (i::t)
| h::i::t -> false

[==
Convertit une Liste_triee.t en liste.
=]
let list_of_t l = l

[==
Convertit une liste en Liste_triee.t
=]
let t_of_list l =
  assert (verifie_triee l);
```

⏪ ⏩ ⏴ ⏵ 🔍

## Moralité

Ce module définit une notion de liste triée (le type `t`)– et la seule manière de créer ou de consulter une valeur de type `t`.

⏪ ⏩ ⏴ ⏵ 🔍

## Moralité

Ce module définit une notion de liste triée (le type `t`)– et la seule manière de créer ou de consulter une valeur de type `t`. Nous nous sommes assurés que personne ne pouvait faire de mal avec notre structure de données (en tout cas tant que les assertions sont actives).

⏪ ⏩ ⏴ ⏵ 🔍

## Moralité

Ce module définit une notion de liste triée (le type `t`) – et la seule manière de créer ou de consulter une valeur de type `t`. Nous nous sommes assurés que personne ne pouvait faire de mal avec notre structure de données (en tout cas tant que les assertions sont activées). Sans entrer dans les détails : on peut encore améliorer les choses en décidant qui a le droit d'employer la fonction `t_of_list`.

## Utilisation

```
ocaml liste_triee.cmo
Objective Caml version 3.09.2
# Liste_triee.t of list::
- : 'a list -> 'a Liste_triee.t = <fun>
```

## Utilisation

```
ocaml liste_triee.cmo
Objective Caml version 3.09.2
# Liste_triee.t of list::
- : 'a list -> 'a Liste_triee.t = <fun>
```

Vous devez charger les fichiers `.cmo`

- ▶ pendant le lancement de OCaml
- ▶ par ordre de dépendances

## Utilisation

```
ocaml liste_triee.cmo
Objective Caml version 3.09.2
# Liste_triee.t of list::
- : 'a list -> 'a Liste_triee.t = <fun>
```

Vous devez charger les fichiers `.cmo`

- ▶ pendant le lancement de OCaml
- ▶ par ordre de dépendances

Avec Camelia, ajoutez les fichiers `.cmo` dans le menu "OCaml", "Configure Camelia", dans "Location of the OCaml Interpreter".

## Ainsi...

```
ocaml liste_triee.cmo
Objective Caml version 3.09.2

# let insere a ~dans:liste =
  let rec aux l = match l with
  | [] -> [a]
  | h::_ when a <= b -> a::l
  | h::T -> h::(aux t)
  in Liste_triee.t_of_list (aux (Liste_triee.list_of_t liste));
val insere : 'a -> dans:'a Liste_triee.t -> 'a Liste_triee.t = <fun>

# let rec trie_par_insertion = function
| [] -> Liste_triee.t_of_list []
| h::t -> insere h ~dans:(trie_par_insertion t);
val trie_par_insertion : 'a list -> 'a Liste_triee.t = <fun>

# Liste_triee.t_of_list [5;4;3;2;1];
Exception: Assert_failed ("liste_triee.ml", 28, 2).

# trie_par_insertion [5;4;3;2;1];
- : int Liste_triee.t = <abstr>
# Liste_triee.list_of_t (trie_par_insertion [5;4;3;2;1]);
- : int list = [1; 2; 3; 4; 5]
```

◀ ▶ ↺ ↻ 🔍

## Compilation...

Un programme OCaml est une liste de types et de valeurs.

◀ ▶ ↺ ↻ 🔍

## Compilation...

Un programme OCaml est une liste de types et de valeurs. D'ailleurs, il n'y a pas de différence entre un programme et un module.

## Compilation...

Un programme OCaml est une liste de types et de valeurs. D'ailleurs, il n'y a pas de différence entre un programme et un module.

```
ocamlc -c liste_triee.ml
ocamlc -c liste_triee.ml
ocamlc -c demo_listes.ml
```

◀ ▶ ↺ ↻ 🔍

◀ ▶ ↺ ↻ 🔍

## ...et liaison

La dernière étape est la liaison : prendre plusieurs modules et en faire un exécutable.

```
ocamlc liste_triee.cmo demo_listes.cmo -o mon_programme
```



C'est bon...

...vous savez tout sur la compilation.



## ...et liaison

La dernière étape est la liaison : prendre plusieurs modules et en faire un exécutable.

```
ocamlc liste_triee.cmo demo_listes.cmo -o mon_programme
```

### Notes

- ▶ L'ordre des modules est important !
- ▶ Certaines bibliothèques sont fournies comme des fichiers .cma et pas .cmo – à vérifier dans la documentation de la bibliothèque.



C'est bon...

...vous savez tout sur la compilation.  
 Maintenant, exercices.



C'est bon...

...vous savez tout sur la compilation.

Maintenant, exercices.

Notez quelque part *toutes* les commandes que vous avez tapées. Vous vous en resservirez plus tard.