▶ **MP & MP* – Option Informatique**
Troisième TP Caml
Mardi 9 et 16 octobre 2007

Types somme et enregistrement

Maintenant que nous savons « parfaitement » utiliser les listes, les références et la programmation impérative en général, nous pouvons voir définir et utiliser de nouveaux types (autres que entier, flottant, chaîne de caractère, liste et vecteur).

Notions : types somme et enregistrement, files d'attente, dérivation formelle

1 Les types sommes

1.1 Définition et utilisation

CamlLight offre directement différents types (int , float , bool, string , list , vect, par exemple), mais parfois ils ne suffisent pas ou ne sont pas utilisables de façon efficace. Cependant, CamlLight offre la possibilité de définir de nouveaux types. On peut par exemple définir un nouveau type qui ne pourra prendre qu'un petit nombre de valeurs différentes.

```
type ternaire = Vrai | Faux | Peutetre;;
Type ternaire defined.
```

▶ **Question 1** *Après avoir défini le type* ternaire *comme précédemment, définissez une fonction* and_t *ou une fonction* or_t *qui respectent les tables de vérité suivantes :*

and_t	*Vrai*	*Faux*	*Peutetre*
Vrai	*Vrai*	*Faux*	*Peutetre*
Faux	*Faux*	*Faux*	*Faux*
Peutetre	*Peutetre*	*Faux*	*Peutetre*

or_t	*Vrai*	*Faux*	*Peutetre*
Vrai	*Vrai*	*Vrai*	*Vrai*
Faux	*Vrai*	*Faux*	*Peutetre*
Peutetre	*Vrai*	*Peutetre*	*Peutetre*

```
and_t : ternaire -> ternaire -> ternaire
or_t : ternaire -> ternaire -> ternaire
```

Une notation infixe sera bien évidemment utilisée pour rendre leur utilisation plus naturelle.
Note *: la fonction inverse de* #infix *"and_t";; est* #uninfix *"and_t";;* .

Mais on peut également définir un type réutilisant un ou plusieurs types existants :

```
type variable = Entier of int | Flottant of float | Bool of bool | Chaine of string;;
Type variable defined.
```

Les types réutilisés (ici int , float , ...) doivent déjà être définis. La syntaxe suivante

```
type variable = int | float | bool | string;;
Type variable defined.
```

est correcte, mais on définit alors un type variable qui ne peut prendre que 4 valeurs.

▶ **Question 2** *Écrivez une fonction* add *qui prend en argument deux variables et renvoie la somme des deux s'il s'agit de flottants ou d'entiers, la concaténation s'il s'agit d'une chaîne et d'une autre variable, le "et" logique s'il s'agit de deux booléens et qui lève une exception sinon.*

```
add : variable -> variable -> variable
```

1.2 Un exemple un peu plus complet : les listes

Les listes peuvent parfaitement être redéfinies en utilisant les types sommes, par exemple avec :

```
type 'a liste = Nil | Cons of 'a * ('a liste);;
```

Note : le ' (« quote ») est en-dessous du 4 sur le clavier.

▶ **Question 3** *Redéfinissez les fonctions élémentaires sur les listes* hd *(renvoie la tête de la liste),* tl *(renvoie la queue de la liste),* rev *(retourne la liste) et* map *(applique une fonction f sur la liste et renvoie les résultats). N'oubliez pas de les tester !*

```
hd2 : 'a liste -> 'a
tl2 : 'a liste -> 'a liste
rev2 : 'a liste -> 'a liste
map2 : ('a -> 'b) -> 'a liste -> 'b liste
```

2 Les types enregistrement

2.1 Définition et utilisation

2.1.1 Principe

Supposons que nous voulions travailler sur la liste des élèves d'une classe et sur leurs notes respectives en mathématiques et en sciences physiques (vous avez le droit de trouver l'exemple bidon). Un moyen simple de représenter ces différentes données serait de construire 3 tableaux différents, un contenant les noms des élèves, les deux autres contenant les notes, respectivement de mathématiques et de sciences physiques. Cependant, cette façon de procéder n'est pas particulièrement pratique ; par exemple, chaque fonction utilisant ces données aura au moins 3 arguments différents.

Une autre façon sûrement plus pratique serait de définir un seul tableau formé de petits tableaux de taille 3 (chacun étant formé du nom de l'élève et des deux notes). Cependant, cette façon, bien que plus jolie, n'est pas réalisable en CamlLight (un tableau contient soit des chaînes, soit des flottants, mais pas un peu des deux).
Fort heureusement, CamlLight nous propose les "types enregistrement" (Record en anglais), qui font exactement ce qui vient d'être décrit.
Un tel type sera défini par :

```
type notes = {nom : string ; maths : float ; physique : float};;
```

Un élève sera alors décrit par :

```
let dupont = {nom ="Dupont" ; maths = −42.0 ; physique = 24.0};;
let dupond = {nom ="Dupond" ; maths = 10.0 ; physique = 3.14};;
```

2.1.2 Premiers exemples

Pour accéder à la valeur d'un champ particulier, par exemple la note en mathématiques, on utilise dupont.maths.

▶ **Question 4** *Ecrivez fonction qui calcule la moyenne d'une classe en mathématiques et physique.*

```
moyenne : notes vect −> float ∗ float
```

Un petit exemple sur le filtrage :

```
let a_15_partout = fun eleve −>
    match eleve with
    | {maths = 15.; physique = 15.} −> true
    | _ −> false;;
```

Note : dans le filtrage, seul le signe = est accepté.
On peut définir une fonction est_bon_eleve par :

```
let est_bon_eleve = fun eleve −>
    match eleve with
    | { maths = m; physique = 0. } −> m >= 19.
    | _ −> eleve.maths >= 15.0;;
```

▶ **Question 5** *Définissez un type* fraction *composé d'un numérateur et d'un dénominateur, entiers tous les deux, et écrivez une fonction* ajoute *qui fasse la somme de deux fractions.*

```
ajoute : fraction −> fraction −> fraction
```

2.1.3 Champs modifiables

Si on modélise un élève par un tableau de flottants (!), on peut modifier ses notes, alors que ce n'est pas le cas lorsqu'on le modélise par un type enregistrement.
Pour changer cela, il suffit de déclarer les champs désirés comme étant modifiables, avec le mot-clé "mutable".

```
type Eleve = {Nom : string; mutable Maths : float; mutable Physique : float};;
let toto = {Nom = "Toto"; Maths = 15.; Physique = 15.};;
toto.Maths <− 16.;;
```

2.2 Modélisation d'une file d'attente

Deux modèles très classiques existent pour les files d'attente : le modèle *LIFO* (Last-in, First-out), ou pile, qui permet l'insertion et la suppression d'éléments en tête, et le modèle *FIFO* (First-in, First-out), ou file, qui permet l'insertion en fin et la suppression en tête.

Nous allons ici nous intéresser à un modèle qui permet l'insertion et la suppression aux deux extrémités de la file. L'idée est d'utiliser deux listes distinctes, une à l'endroit qui représente le début, et une à l'envers qui représente la fin. Comme il est facile d'ajouter un élément en tête de liste ou d'en supprimer, la liste est bien adaptée. Si jamais on veut récupérer la tête de notre file et que la liste de début est vide, on retourne la liste de fin et on la met à la place de la liste de tête.

▶ **Question 6** *Définissez un type* file *qui corresponde au modèle défini ci-dessus, ainsi qu'une variable* nil *pour la file vide.*

▶ **Question 7** *Écrivez les fonctions d'insertion et de suppression en tête et en queue. Pour la suppression, il faut renvoyer l'élément enlevé et la file ainsi obtenue alors que pour l'insertion il faudra renvoyer la nouvelle file.*

```
ajoute_tete : 'a file −> 'a −> 'a file
ajoute_queue : 'a file −> 'a −> 'a file
retire_tete : 'a file −> 'a ∗ 'a file
retire_queue : 'a file −> 'a ∗ 'a file
```

3 Calculons !

Nous allons nous intéresser maintenant au fonctionnement d'une calculatrice.
Les calculatrices actuelles utilisent un arbre pour les calculs en interne, qu'elles parcourent en profondeur d'abord.

Nous allons d'abord étudier le cas de la calculatrice 4 opérations.

▶ **Question 8** *Définissez un type* Expr *qui permet de représenter une addition/soustraction/multiplication/-division de deux* Expr*, la négation d'une* Expr *ou un entier. Donnez l'expression correspondant à* $2*(4+(-2))$.

▶ **Question 9** *Grâce à ce type, nous allons pouvoir parcourir récursivement l'arbre qui représente l'expression. Ecrivez une fonction récursive* calcule *qui simplifie l'expression donnée en argument pour renvoyer une seule valeur entière.*

```
calcule : Expr −> int
```

Maintenant, intéressons-nous (sisi) à un type de calcul plus compliqué, la dérivation formelle. Comme précédemment, nous supposerons que l'expression à dériver nous est fournie sous forme d'arbre.

▶ **Question 10** *Définissez un type* formel *qui permette de représenter une expression contenant les* 4 *opérations de base, des entiers, des flottants, l'élévation à une puissance entière, diverses fonctions (*sin*, ...), des variables (x, y, ...). Pour ne pas avoir une définition trop longue, les différentes opérations binaires seront notées sous forme de chaîne de caractère, de même que les nom de variables.*

▶ **Question 11** *En vous inspirant de la fonction précédente, écrivez une fonction qui dérive une expression formelle par rapport une variable donnée. On n'essaiera pas de simplifier le résultat obtenu.*

```
derive : formel -> string -> formel
```

▶ **MP & MP* – Option Informatique**
Troisième TP Caml
Mardi 9 et 16 octobre 2007

Types somme et enregistrement

Un corrigé

▶ Question 1

```
type ternaire = Vrai | Faux | Peutetre;;
#uninfix "and_t";;
let and_t = fun a b ->
    match (a,b) with
    | (Vrai, Vrai) -> Vrai
    | (Faux, _) -> Faux
    | (_ , Faux) -> Faux
    | (_ , Peutetre) -> Peutetre
    | (Peutetre, _) -> Peutetre
;;
#infix "and_t";;
Vrai and_t Peutetre;;


#uninfix "or_t";;
let or_t = fun a b ->
    match (a,b) with
    | (Vrai, _) -> Vrai
    | (_ , Vrai) -> Vrai
    | (Faux, Faux) -> Faux
    | (_ , _) -> Peutetre
;;
#infix "or_t";;
Faux or_t Peutetre;;
```

▶ Question 2

```
let add = fun x y ->
    match (x,y) with
    | (Entier x, Entier y) -> Entier (x + y)
    | (Flottant x, Flottant y) -> Flottant (x +. y)
    | (Flottant x, Entier y) -> Flottant (x +. float_of_int y)
    | (Entier x, Flottant y) -> Flottant ((float_of_int x) +. y)
    | (Bool x, Bool y) -> Bool (x && y)
    | (Chaine x, Entier y) -> Chaine (x ^ string_of_int y)
    | (Chaine x, Flottant y) -> Chaine (x ^ string_of_float y)
    | (Chaine x, Bool y) -> Chaine (x ^ string_of_bool y)
    | (Chaine x, Chaine y) -> Chaine (x ^ y)
    | (Entier x, Chaine y) -> Chaine (string_of_int x ^ y)
    | (Flottant x, Chaine y) -> Chaine (string_of_float x ^ y)
    | (Bool x, Chaine y) -> Chaine (string_of_bool x ^ y)
    | (_ , _) -> failwith "types_incompatibles";;

add (Entier 42) (Entier 5);;
add (Entier 42) (Flottant 3.1415);;
add (Chaine "plop_") (Entier 42);;
add (Bool true) (Entier 5);;
```

▶ Question 3

```
let hd2 = fun liste ->
    match liste with
    | Nil -> failwith "liste_vide"
    | Cons (a, b) -> a;;
let tl2 = fun liste ->
    match liste with
    | Nil -> failwith "liste_vide"
    | Cons (_, a) -> a;;

let rev2 = fun liste -> (
    let rec aux = fun debut fin ->
     match debut with
      | Nil -> fin
      | Cons(t, q) -> aux q (Cons(t,fin))
    in aux liste Nil);;

let rec map2 = fun f liste ->
    match liste with
    | Nil -> Nil
    | Cons(t , q) -> Cons((f t), (map2 f q));;

hd2 Nil;;
let test = Cons (1, Cons (2, Cons (3, Nil)));;
hd2 test;;
tl2 test;;
rev2 test;;
map2 (fun x -> x*x) test;;
```

▶ Question 4

```
let moyenne = fun eleves ->
    let n = vect_length eleves and maths = ref 0. and physique = ref 0. in
    for i = 0 to (n-1) do (
        maths := !maths +. (eleves.(i)).maths;
         physique := !physique +. (eleves.(i)).physique;
    )
    done;
    (!maths /.(float_of_int n), (!physique) /. (float_of_int n))
;;
moyenne [|dupont ; dupond |];;
```

▶ Question 5

```
type fraction = {num : int; denum : int};;
let ajoute = fun x y ->
    { num = x.num * y.denum + y.num * x.denum;
    denum = x.denum * y.denum};;
```

▶ Question 6

```
type 'a file = {tete : 'a list; queue : 'a list };;
let nil = { tete = []; queue = [] };;
```

▶ Question 7

```
let ajoute_tete = fun {tete = a; queue = b} elt ->
    {tete = elt::a ; queue = b};;
let ajoute_queue = fun {tete = a; queue = b} elt ->
    {tete = a ; queue = elt::b};;
let retire_tete = fun {tete = a; queue = b} ->
    match (a, b) with
    | ([] , []) -> failwith "file_vide"
    | ([] , t::q) -> (t, {tete = (rev q); queue = []} )
    | (t::q, b) -> (t, {tete = q; queue = b});;
let retire_queue = fun {tete = a; queue = b} ->
    match (a, b) with
    | ([] , []) -> failwith "file_vide"
    | (t::q , []) -> (t, {tete = []; queue = (rev q)} )
    | (a , t::q) -> (t, {tete = a; queue = q});;
retire_queue {tete = [1; 2; 3]; queue = [4; 5; 6]};;
retire_tete {tete = [1; 2; 3]; queue = [4; 5; 6]};;
```

▶ Question 8

```
type Expr = Plus of Expr * Expr
    | Moins of Expr * Expr
    | Fois of Expr * Expr
    | Div of Expr * Expr
    | Unaire of Expr
    | Val of int;;
(Fois(Val 2, Plus (Val 4, (Unaire (Val 2)))));;
```

▶ Question 9

```
let rec calcule = fun expression ->
    match expression with
    | Val a -> a
    | Plus(a,b) -> (calcule a) + (calcule b)
    | Moins(a,b) -> (calcule a) - (calcule b)
    | Fois(a,b) -> (calcule a) * (calcule b)
    | Div(a,b) -> (calcule a) / (calcule b)
    | Unaire a -> -(calcule a)
;;
calcule (Fois(Val 2, Plus (Val 4, (Unaire (Val 2)))));;
```

▶ Question 10

```
type formel = binaire of string * formel * formel | unaire of string * formel | variable of string | entier of int | reel of float;;
```

▶ Question 11

```
let rec derive = fun expression var ->
    match expression with
    | reel a -> reel 0.
    | entier a -> entier 0
    | variable a when a = var -> entier 1
    | variable a -> entier 0
    | unaire ("-", x) -> unaire("-", (derive x var))
    | unaire ("sin", x) -> binaire("*", (derive x var), unaire("cos", x))
    | unaire ("cos", x) -> unaire("-", binaire("*", (derive x var), unaire("sin", x)))
    | binaire ("+",x,y) -> binaire("+" , (derive x var) , (derive y var))
    | binaire ("-",x,y) -> binaire("-" , (derive x var) , (derive y var))
    | binaire ("*",x,y) -> binaire("+", binaire("*" , x , (derive y var)), binaire("*" , (derive x var) , y))
    | binaire ("/", x, y) -> binaire("/",
     binaire("-", binaire("*", (derive x var) , y) , binaire("*", x , (derive y var)) ),
     binaire("^", y, entier 2))
    | binaire ("^", x, entier y) -> binaire("*", binaire("*", entier y, (derive x var)), binaire("^", x, entier (y - 1)))
    | _ -> failwith "opération_inconnue"
      ;;

    derive (binaire("+", binaire("^", variable "x", entier 2), unaire("sin", variable "x"))) "x";;
```