Fonctions à un paramètre

Syntaxe

Voici un exemple de définition d'une fonction :

```
let abs x =
  if x<0 then -x
  else x;;
```

La traduction française est : « Soit `abs` une fonction telle que `abs` appliqué à x vaut –x si x est négatif et x sinon. » Quand on compile cette définition de fonction, le message suivant s'affiche :

```
abs : int -> int = <fun>
```

Il signifie que `abs` est une fonction qui à un entier (`int`) associe un entier. Ces types sont inférés automatiquement par le compilateur. Essayons d'appliquer cette fonction à différentes valeurs :

`abs 3;;`	le résultat est 3.
`abs (3);;`	le résultat est 3.
`abs -5;;`	un message d'erreur apparait.
`abs (-5);;`	le résultat est 5.

L'erreur du troisième essai est due au fait que CAML interprète abs -5 comme « (abs appliqué à -) appliqué à 5. » Les priorités des opérateurs n'est pas la même que dans d'autres langages de programmation !

Si maintenant on tente d'exécuter `abs 3.8`, un message d'erreur apparait car 3.8 n'est pas un entier. Pour réaliser une fonction de calcul de valeur absolue d'un flottant, on peut écrire :

```
let abs x =
  if x<0. then -.x
  else x;;
```

En CAML, les opérateurs sur les nombres flottants ont une syntaxe différente de ceux sur les entiers. C'est nécessaire pour que l'inférence de types puisse fonctionner. Les opérateurs de base sur les entiers sont `– + * / mod` et ceux sur les flottants sont `-. +. *. /.` Il existe des opérateurs de conversion : `float_of_int` converti un flottant en entier et `int_of_float` convertit un entier en flottant.

> Donnez la définition d'une fonction récursive `pow2` telle que `pow2` appliquée à x calcule 2 à la puissance x.

Curryfication

Regardez la définition suivante :

```
let sum a b = a + b;;
```

La fonction `sum` ne possède pas deux paramètres, mais un seul, nommé `a`, et elle retourne une fonction qui appliquée à `b` retourne `a+b`. En français on pourrait dire « (`sum` appliqué à `a`) appliqué à `b` vaut `a+b` ». Quand on valide cette définition, le compilateur répond :

```
sum : int -> int -> int = <fun>
```

Cette signature doit s'interpréter avec la règle d'associativité à droite des abstractions du lamda-calcul :

```
sum : int -> (int -> int) = <fun>
```

Autrement dit, `sum` est une fonction qui à un entier associe une fonction qui à un entier associe un entier. Il est important de bien comprendre cette notion qui permet de simuler une fonction à plusieurs paramètres à l'aide de fonctions acceptant un seul paramètre, et qui est appelée curryfication. Analysons les implications pratiques sur des exemples d'appel de la fonction.

`sum 10 15 ;;`	retourne `int=25`.
`(sum 10) 15 ;;`	retourne `int=25`.
`sum (10 15) ;;`	retourne un message d'erreur car (10 15) n'est pas un entier !

Les première et deuxième lignes sont équivalentes, les parenthèses étant implicites dans la première du fait de l'associativité à gauche des applications, qui est héritée du lambda-calcul. La troisième ligne est incorrecte parce que (10 15) n'est pas un entier mais l'application de 10 à 15 qui ne veut rien dire en CAML. Examinons d'autres exemples.

`sum 10 ;;`	retourne `int -> int = <fun>`.
`let f=sum 10 ;;`	retourne `f : int -> int = <fun>`.
`f 15 ;;`	retourne `int=25`.

Si on applique sum à 10, le résultat est une fonction qui à un entier `b` associe cet entier `b+10`. Mais cette fonction est perdue. Dans la deuxième ligne, on définit cette fonction en lui donnant l'identifiant f, ce qui permet de la réutiliser dans la troisième ligne.

> Soit la fonction f définie de la manière suivante : `let f a b c = a+(b*c);;`
> Donnez la signature de `f`. Réécrivez l'expression `f 10 20 30` avec des parenthèses.
> Si on écrit : `let g = f 10 ;;` quel est la signature de `g` et que réalise cette fonction `g` ?
> Est-ce que `let h = g 20 ;;` est strictement équivalent à `let h = f 10 20 ;;` ?

Remarque1 : il existe une autre syntaxe pour la définition d'une fonction. Pour définir la fonction `sum` avec cette autre syntaxe, il faut écrire :

```
let sum = fun a -> fun b -> a+b;;
```

Qui peut se lire « soit `sum` la fonction qui à `a` associe la fonction qui à `b` associe `a+b` ». Dans la suite, nous n'utiliserons pas cette syntaxe.

Remarque 2 : CAML peut aussi gérer des couples, triplets, n-uplets de valeurs. Il est donc possible de définir une fonction `add` de la manière suivante :

```
let add (a,b) = a+b;;
```

Cette fonction `add` permet bien de calculer la somme de deux entiers mais n'est pas équivalente à la fonction `sum`. Sa signature est :

```
add : int * int -> int = <fun>
```

Ce qui signifie que `add` est une fonction qui appliquée à une couple d'entiers (`int * int`) associe un entier. Elle ne s'utilise pas de la même manière que `sum` :

`sum (10,15);;`	retourne `int = 25`.
`sum (10 15);;`	produit un message d'erreur.
`sum  10 15;;`	produit un message d'erreur.

Cette approche est moins dans l'esprit de CAML que la curryfication. Nous éviterons donc de l'utiliser.

Listes

Une liste peut être construite ou décomposée avec [] et :: comme le montrent les exemples suivants.

```
let m = 1 :: 2 :: 3 :: [];;
let p = 4 :: m;;
```

Pour lesquels CAML répond :

```
m : int list = [1; 2; 3]
p : int list = [4; 1; 2; 3]
```

La reconnaissance de motifs (pattern matching) permet de simplifier les programmes exploitant des listes. Par exemple, la fonction définie ci-dessous calcule la longueur d'une liste.

```
let rec longueur w =
  match w with
  | [] -> 0;
  | x::q -> 1 + longueur q;;
```

Sa signature est : longueur : `'a list -> int = <fun>`, qui s'interprète comme « une fonction qui a une liste d'élément de type 'a quelconque associe un entier ». Voici un exemple d'application de cette fonction :

`longueur [1;2;7];;` retourne `int = 3`.

Complétez le code de cette fonction qui doit retourner le dernier élément d'une liste :

```
let rec dernier w =
  match w with
   |[] -> failwith "Fonction non définie pour une liste vide"
   |[e] -> ______
   |x :: q -> ____________________ ;;
```

Donnez la signature de cette fonction.

Complétez le code de la fonction insert de manière à ce que appliquée à un élément e et une liste triée w, elle retourne la liste obtenue en insérant e dans w à la bonne place :

```
let rec insert e w =
  match w with
   | [] -> __________
   | x :: q ->
      if e < x then ____________________
      else x :: ________________________;;
```

Donnez la signature de la fonction `insert`.

Fonctions d'ordre supérieur

Une fonction d'ordre supérieur est une fonction acceptant une fonction en paramètre. Cela ne pose absolument aucun problème en CAML. Par exemple, la fonction sigma définie ci-dessous applique une fonction f à tous les éléments d'une liste w et retourne la somme des valeurs obtenues.

```
let rec sigma f w =
  match w with
   | [] -> 0
   | x :: q -> f x + sigma f q;;
```

Sa signature est sigma : `('a -> int) -> 'a list -> int = <fun>` pour « une fonction qui (à toute fonction associant à une valeur de type 'a quelconque un entier) associe (une fonction qui à toute liste d'éléments de type 'a associe un entier).

Donc par exemple :

`let h x = x*x;;`	définit une fonction `h` qui calcule le carré d'un entier.
`sigma h [1;2;3;4];;`	retourne `(f 1) + (f 2) + (f 3) + (f 5)`, soit 30.

Donnez la signature de `sigma h` (i.e. le résultat de `sigma` appliqué à `f`).

Donnez la définition d'une fonction `applique` telle que si `f` est une fonction de type `'a -> 'a` et `w` une liste d'éléments de type `'a` (où `'a` est un type quelconque), alors `applique f w` retourne la liste des valeurs obtenues en appliquant `f` à tous les éléments de `w`.
Par exemple, avec la fonction `h` décrite ci-dessus calculant le carré d'un entier, `applique h [1;2;3;4];;` doit retourner `[1;4;9;16]`.

Variantes et applications aux arbres

Une variante est un type dont les éléments peuvent prendre plusieurs formes. Par exemple :

```
type btree = E | N of int * btree * btree ;;
```

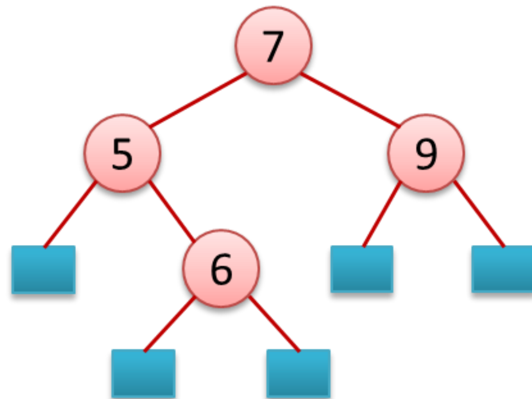
Signifie que les éléments (ou valeurs) du type `btree` sont de deux sortes :

- l'élément E, qui représente un nœud vide, où feuille d'un arbre,
- des éléments de la forme N `(x,g,d)`, où x est un entier et où `g` et `d` sont des arbres de type `btree` que nous appellerons fils gauche et fils droit de l'arbre N `(x,g,d)`.

Les lettres E (comme Empty) et N (come Node) sont arbitraires. On aurait pu choisir d'autres lettres ou mots commençant par une majuscule.
La notation `N of int * btree * btree` est la manière de spécifier un triplet de la forme `N(int,btree,btree)`. Cette syntaxe n'est pas intuitive pour tout le monde, donc il est important que vous vous l'appropriiez.

Voici un exemple d'arbre spécifié à l'aide du type btree :

```
let t =
N (7,N (5,E,N(6,E,E)),N (9,E,E));;
```



Dessinez l'arbre spécifié par le terme `N(1,E,N(7,N(5,E,E),E))`.

Définissez un type permettant de représenter des arbres binaires dans lesquels les nœuds ne sont pas étiquetés par des entiers (ils ont simplement deux fils), mais où chaque feuille, par contre, est étiquetée par un entier.

La fonction suivante vérifie si un entier se trouve quelque part dans un arbre du type btree défini au début de cette fiche.

```
let rec member x t =
  match t with
   |E -> false
   |N(y, left, right) ->
        if x=y then true
        else if member x left then true
        else member x right;;
```

Améliorez la définition de la fonction `member` ci-dessus de manière à la rendre plus efficace dans le cas où l'arbre `t` est trié, c'est-à-dire que dans un arbre `N(x,g,d)`, toutes les valeurs situées dans `g` sont inférieures ou égales à `x` et toutes celles situées dans `d` sont supérieures à `x`.