Institut National Universitaire Champollion

TP 7 - Les types sommes

Programmation fonctionnelle en Caml
L3 INFO - Semestre 6

Nous allons illustrer au cours de ce TP les différentes formes de types sommes :

- Les types sommes par énumération de constantes,
- Les types sommes construits
- Les types sommes récursifs

1 - Les couleurs

1. Définir un type énuméré *couleur* contenant les constantes : Blanc, Noir, Rouge, Vert, Bleu, Jaune, Cyan et Magenta.
2. Écrire une fonction `est_colore : couleur -> bool` qui renvoie faux pour Noir et Blanc, et vrai pour les autres couleurs du type `couleur`.
3. Écrire une fonction `complementaire : couleur -> couleur` qui, à une couleur, associe son complémentaire.

2 - Nombres entiers, réels et complexes

1. Définir un type somme `nombreNR`, regroupant les nombres entiers et réels.
2. Écrire des fonctions `somme : nombreNR * nombreNR -> nombreNR` et `prod : nombreNR * nombreNR -> nombreNR` mettant en œuvre l'addition et la multiplication de deux objets de type `nombreNR`.
3. Définir un type somme `nombreRC`, regroupant les nombres réels et complexes (représentés par le couple de réels (partie réelle, partie imaginaire)).
4. Écrire des fonctions `somme : nombreRC * nombreRC -> nombreRC` et `prod : nombreRC * nombreRC -> nombreRC` mettant en œuvre l'addition et la multiplication de deux objets de type `nombreRC`.

3 - Un premier type somme récursif : Les expressions logiques

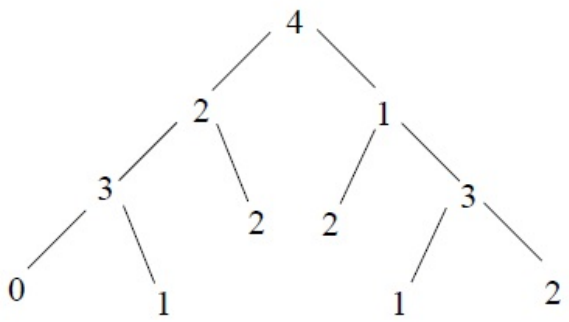
Une expression logique peut être féinie à l'aide du type somme énuméré suivant :

```
type exprLogique =
Vrai | Faux
| Non of exprLogique
| Et of exprLogique*exprLogique
| Ou of exprLogique*exprLogique ; ;
```

1. Définir l'expression logique *exp = ((V et V) ou (non (F ou non V)) et V)*.
2. Écrire une fonction `echange : exprLogique -> exprLogique` qui, au sein d'une expression logique, échange toutes les constantes `Vrai` et `Faux`.
3. Écrire une fonction `evalue : exprLogique -> bool` qui calcule la valeur d'une expression logique. Que vaut l'expression *exp* ?

4 - Les arbres d'entiers

Considérons des arbres binaires dont les nœuds et les feuilles sont des entiers. Par exemple, appelons T l'arbre suivant :



Un tel arbre peut être décrit par le type récursif suivant :

```
type arbre =  Feuille of int
| Noeud of int*arbre*arbre ; ;
```

L'arbre T peut alors être défini par :

```
let t= Noeud  (4,   Noeud    (2, Noeud (3, Feuille 0, Feuille 1),  Feuille 2),
Noeud    (1,  Feuille 2, Noeud (3, Feuille 1, Feuille 2))) ; ;
```

1. Écrire une fonction `profondeur : arbre -> int` calculant la profondeur d'un arbre, c'est à dire, la plus grande distance entre la racine et une feuille de l'arbre. La hauteur de l'arbre T est 3. On écrire localement la fonction `max` qui calcule le maximum de deux entiers.
2. Un arbre binaire est dit **complet de profondeur** p si toutes ses feuilles sont à la même profondeur p. T n'est pas complet car il possède 6 feuilles, 2 de profondeur 2 et 4 de profondeur 3.

   Écrire une fonction `complet : int * int -> arbre` qui, à un entier p et un entier n, associe l'arbre binaire complet de profondeur p dont tous les nœuds et toutes les feuilles valent n.

   ```
   complet(3,1) ; ;
   -  : arbre = Noeud  (1,   Noeud    (1, Noeud (1, Feuille 1, Feuille 1), Noeud (1, Fe
   Noeud (1, Noeud (1, Feuille 1, Feuille 1), Noeud (1, Feuille 1, Feuille 1)))
   ```

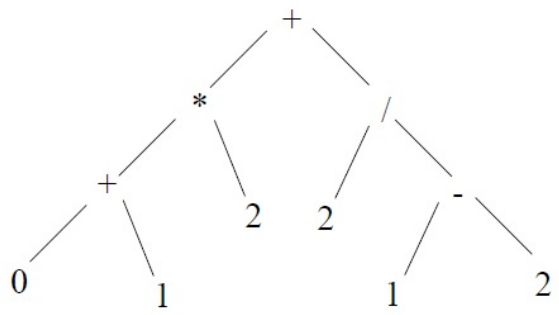
3. Écrire une fonction `liste : arbre -> int list` qui, à un arbre, associe la liste des valeurs de ses nœuds et de ses feuilles en donnant la valeur du nœud, puis le résultat du parcours du sous-arbre gauche, puis celui du sous-arbre droit :

   ```
   liste(t) ; ;
   -  : int list = [4 ; 2 ; 3 ; 0 ; 1 ; 2 ; 1 ; 2 ; 3 ; 1 ; 2]
   ```

   Expliciter les différents appels récursifs lors de l'appel à `liste(t)`.

5 - Les expressions arithmétiques

Considérons maintenant des arbres binaires dont les nœuds sont des opérations et les feuilles sont des entiers. Un tel arbre est utilisé pur représenter une expression arithmétique.
Par exemple, l'expression arithmétique $exp = ((0+1)*2)+(2/(1-2))$ peut être représentée par l'arbre T ci-dessous :



En Caml, nous utiliserons le type récursif suivant :

```
type expArithm =  Nb of int
| Oper of char*expArithm *expArithm  ; ;
```

1. Écrire l'expression exp.
2. Écrire une fonction `nbOper : expArithm -> int` qui compte le nombre d'opérations contenues dans une expression arithmétique.
3. Écrire une fonction `evalue : expArithm -> int` qui calcule la valeur d'une expression arithmétique. On pourra écrire au préalable une fonction `calcul : char * int * int -> int` définie par filtrage sur le caractère c et mettant en œuvre le calcul d'une des quatre opérations entre deux entiers a et b.

6 - Les mobiles

Le but de cet exercice est d'écrire un certain nombre de fonctions aidant à la réalisation de mobiles décoratifs pour berceaux.

Un mobile est constitué de baguettes de bois horizontales suspendues par des fils verticaux et aux extrémités desquels pendent des objets.

Les objets suspendus aux extrémités des tiges seront décrits par un type `objet` et constitueront les feuilles du mobile.

```
type  objet = Chat | Clown | Mouton ; ;

type mobile = Feuille of objet | Noeud of (int*int*mobile*mobile) ; ;
```

Les entiers représentent les longueurs des tiges. Nous associons à chaque objet un poids :

```
let poids_f =  fun
Chat -> 1
| Clown -> 3
| Mouton -> 2  ; ;

poids_f  : objet -> int = <fun>
```

On suppose que le poids des fils et des tiges est négligeable.

1. Écrire une fonction à deux arguments `fait_mob_simple : int * objet -> mobile` permettant de construire un mobile constitué de deux tiges de même longueur donnée et de deux objets identiques à un objet donné.

```
let m1 = fait_mob_simple(2,Chat) ; ;
- m1  : mobile = Noeud (2, 2, Feuille Chat, Feuille Chat)
```

2. Écrire une fonction `poids_m : mobile -> int` permettant de calculer le poids total d'un mobile, c'est à dire la somme des poids de tous les objets d'un mobile.
3. Écrire une fonction `agrandir_m : mobile -> mobile` permettant de construire un mobile en doublant la longueur de toutes les tides d'un mobile donné.
4. Écrire une fonction `echanger_m : objet * objet * mobile -> mobile` permettant de construire un mobile en échangeant deux objets à partir d'un mobile donné.

```
let m2 = Noeud(2,4, Feuille(Mouton),
Noeud (3,4, Feuille (Chat), Feuille (Clown))) ; ;

echanger_m (Chat, Mouton, m2) ; ;
-  : mobile = Noeud (2,4, Feuille(Chat) ,
Noeud (3,4, Feuille(Mouton), Feuille (Clown)))
```

5. Un constructeur souhaite réaliser ce type de mobiles. Il a besoin de connaître la longueur des tiges utilisées, les différentes longueurs utilisées et les différents objets utilisés.
   a) Écrire une fonction `tiges_m : mobile -> int` permettant de calculer la longueur totale des tiges d'un mobile.
   b) Écrire une fonction `lg_tiges_m : mobile -> int list` permettant de calculer la liste des longueurs de toutes les tiges d'un mobile.
   c) Écrire une fonction `objets_m : mobile -> objet list` permettant de connaître la liste de tous les objets d'un mobile.
   Par exemple, `(objets_m m1)` retourne la liste [$Chat$,$Chat$] et `(objets_m m2)` retourne la liste [$Mouton$, $Chat$, $Clown$].
6. Un mobile est dit **localement équilibré** s'il est réduit à une feuille, OU s'il est de la forme $Noeud(\ell_1,\ell_2,m_1,m_2)$ avec $\ell_1 \times$ `poids_m` $m_1 = \ell_2 \times$ `poids_m` m_2.

   Un mobile est dit **globalement équilibré** si tous ses nœuds sont localement équilibrés.
   a) Écrire une fonction `eq_loc : mobile -> bool` permettant de déterminer si un mobile est localement équilibré.
   b) Écrire une fonction `eq_glob : mobile -> bool` permettant de déterminer si un mobile est globalement équilibré.
7. On appelle `profondeur d'une feuille`, la somme des longueurs des tiges à suivre pour atteindre cette feuille. Dans le mobile `m2`, la feuille `Chat` est à une profondeur 7.

   On appelle `profondeur d'un mobile`, la profondeur maximale d'une feuille de ce mobile (c'est à dire, la profondeur de la feuille la plus profonde).
   Écrire une fonction `profondeur : mobile -> int` permettant de calculer la profondeur d'un mobile. Par exemple, `profondeur m2` doit renvoyer 8.

8. Écrire la fonction `nombre_objets : mobile -> int list` qui détermine le nombre d'objets de chaque catégorie d'un mobile donné.
9. Écrire la fonction `egale_m : mobile * mobile -> bool` qui détermine si deux mobiles sont égaux.
10. Écrire la fonction `hauteur : mobile -> int` qui calcule le nombre maximum de fils verticaux dans un mobile donné. Par exemple, le mobile `m2` a pour hauteur 3.

    Nous dirons qu'un mobile m_1 est un sous-mobile du mobile m_2 si m_1 et m_2 sont égaux ou si m_1 est égal à un mobile participant à la construction de m_2.
11. Écrire une fonction `sous_mobile : mobile * mobile -> bool` qui détermine si un mobile est un sous-mobile d'un mobile donné.
12. Écrire la fonction `remplacer : mobile * mobile -> mobile` qui, dans un mobile, remplace toutes les occurrences d'un sous-mobile par un mobile donné.