Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 3
Dessin d'arbres n-aires

Travailler sur les arbres c'est bien... encore faut-il pouvoir les visualiser. On considère ici des arbres n-aires définis par[1] :

```
type 'a arbre = Noeud of 'a * 'a arbre list
```

et on aimerait écrire un algorithme de calcul de la position dans le plan euclidien de chacun des noeuds de l'arbre de telle sorte que le dessin qui en résulte soit *joli*. La difficulté est de définir ce qu'est un "joli arbre".

Dans tous les cas on souhaitera bien entendu que la géométrie du dessin corresponde à celle de l'arbre, et qu'il n'y ait pas croisement de branches... il est clair également que deux noeuds de même profondeur devront avoir la même ordonnée, et on imposera une différence constante des ordonnées de deux niveaux consécutifs. Il reste donc à déterminer les abscisses de chaque noeud... dans ce TP, ces abscisses seront représentées de manière relative (la racine aura une position de `0.`) et seront conservées dans une structure d'"arbre placé" :

```
type 'a arbre_place = NoeudPlace of 'a * float * 'a arbre_place list
```

Votre but va être d'écrire une fonction `x_ifie` qui décore un `arbre` initial avec de "jolies" abscisses pour donner un `arbre_place`.

Récupérez le fichier `TP3.ml` qui contient les définitions de type, quelques arbres exemples, ainsi qu'une fonction `dessine : arbre_place -> unit` qui dessine un arbre déjà placé.

1 Le partage aveugle

La première exigence que l'on peut avoir correspond au "centrage" relatif de chaque noeud : l'abscisse d'un noeud père sera égale à la moyenne des abscisses de ses fils, et ceux-ci seront équidistants. On peut tout d'abord partager la largeur dont dispose chaque noeud en autant de parts qu'il a de fils.

Supposons qu'en un noeud `Noeud(n,[a1;a2;a3])` on ait déjà placé de manière récursive les sous-arbres `a1`, `a2` et `a3`. Il faut maintenant placer ces sous-arbres "autour" de la position `0.`.

a. Écrire une fonction `place : float -> 'a arbre_place list -> 'a arbre_place list` telle que, si `l` est une liste d'arbres placés (les fils d'un noeud), `place delta l` renvoie la même liste où

[1] Une feuille sera donc un noeud sans fils, et, dans ce modèle, il n'existe pas d'arbre vide

la racine de chaque arbre est placée de manière équidistante aux autres, l'ensemble étant centré autour de la position de la racine `0.`. On pourra tout d'abord écrire une fonction auxiliaire `place_aux` prenant un argument supplémentaire représentant la position du fils le plus à gauche.

b. Écrire une fonction récursive `x_ifie : float -> 'a arbre -> 'a arbre_place` qui prend la largeur totale de l'arbre et effectue le placement. Quelle est sa complexité ? Que pensez-vous du résultat ?

2 Les franges

Nous allons ici utiliser une technique plus efficace pour dessiner les arbres en utilisant les franges : une frange est une liste de positions représentant le "bord" d'un arbre. Chaque arbre placé peut être ainsi accompagné de ses franges gauche et droite.

```
type 'a arbre_avec_franges = Franges of float list * 'a arbre_place * float list
```

c. Écrire une fonction `distance_franges fd fg` qui, étant données deux franges se faisant face, calcule la distance minimale des racines pour que les deux arbres soient à distance d'au moins `1.`.

d. On désire cependant garder l'équidistance entre tous les fils... Écrire une fonction `trouve_delta` qui prend en argument une liste d'arbres avec franges et renvoie un `float` permettant de respecter les `distance_frange` déterminées deux à deux sur les sous-arbres.

Nous savons maintenant calculer l'espacement optimal entre les fils d'un noeud. Il reste à écrire les fonctions permettant de construire récursivement un arbre placé avec franges à partir de ses fils.

e. Adapter la fonction `place` de la première partie pour qu'elle décale aussi les franges.

f. Écrire les fonctions `fusionne_franges_gauches` et `fusionne_franges_droites` qui prennent une liste d'arbres avec franges et renvoie les franges de l'arbre complet. On pourra d'abord réaliser une fonction `complete_frange f1 f2` qui complète la frange `f1` par la frange `f2` dans le cas où `f2` est plus longue.

g. Réaliser enfin une fonction `x_ifie : 'a arbre -> 'a arbre_avec_franges` réalisant le placement total de l'arbre. Admirer le résultat.

h.* Quelle est la complexité de cet algorithme ?

Sources

Ce TP a été inspiré par un article de Laurent Chéno dans la *Lettre de Caml* numéro 5 (1996) dont j'ai recopié certaines parties. Cette lettre est disponible à partir du site `http://caml.inria.fr/`.