Option informatique
deuxième épreuve

Lycée Louis-le-Grand, Paris

mars 2000

1 Arbres et dictionnaires

On considère l'alphabet $\mathcal{A} = \{\texttt{G}, \texttt{D}\}$. On représentera un mot sur cet alphabet par une liste de ses caractères, et, dans toute la suite, on pourra supposer qu'on a procédé aux déclarations suivantes :

```
type Char = G | D ;;
type mot == Char list ;;
```

On utilisera les notations habituelles sur les mots (même s'ils sont ici représentés par des listes) : si c est un caractère ($c \in \{\texttt{G}, \texttt{D}\}$), et si m est un mot, on notera $c.m$ le mot obtenu par concaténation ; plus généralement, si X est un ensemble de mots, $c.X$ désignera l'ensemble $\{c.x, x \in X\}$.
On se propose d'étudier une structure de données informatique qui permette de gérer un *dictionnaire*, c'est-à-dire un ensemble $\mathcal{E}$ de mots sur l'alphabet $\mathcal{A}$.
On choisit d'utiliser une structure d'arbre binaire définie par le type suivant :

```
type arbre =
    | Vide
    | Feuille
    | Nœud of arbre * arbre
    | NœudFeuille of arbre * arbre ;;
```

On définit une sémantique sur ces arbres en définissant une application μ de l'ensemble de ces arbres dans l'ensemble des mots sur l'alphabet $\mathcal{A}$ de la façon suivante :

$$\begin{aligned}
\mu(\texttt{Vide}) &= \emptyset\,; \\
\mu(\texttt{Feuille}) &= \{\varepsilon\}, \quad \text{où } \varepsilon \text{ représente le mot vide}\,; \\
\mu(\texttt{Nœud(g,d)}) &= \texttt{G}.\mu(\texttt{g}) \cup \texttt{D}.\mu(\texttt{d})\,; \\
\mu(\texttt{NœudFeuille(g,d)}) &= \{\varepsilon\} \cup \texttt{G}.\mu(\texttt{g}) \cup \texttt{D}.\mu(\texttt{d})
\end{aligned}$$

Ainsi, une descente à gauche dans un arbre *préfixe par `G`* les mots que représente le sous-arbre gauche, et une descente à droite dans un arbre *préfixe par `D`* les mots que représente le sous-arbre droit.
Les nœuds du genre `NœudFeuille` permettent de signaler un mot intermédiaire.
Par exemple, le dictionnaire $\mathcal{E} = \{\texttt{G}, \texttt{GGDGD}, \texttt{GDDG}, \texttt{GDGG}, \texttt{GD}, \texttt{DG}, \texttt{DGD}\}$ est-il représenté par l'arbre de la figure 1 page suivante, où chaque carré représente une `Feuille` ou un `NœudFeuille`, tandis qu'un rond correspond à un `Nœud`.
De tels arbres seront dorénavant appelés *arbres-dictionnaires*.

1.1 Profondeur et taille

1.1.1 Profondeur

Écrire une fonction `profondeur : arbre -> int` qui renvoie la profondeur d'un arbre-dictionnaire. Par exemple, dans le cas de l'arbre de la figure 1 page suivante, cette fonction doit renvoyer l'entier `5`. On conviendra de renvoyer -1 dans le cas d'un arbre vide.

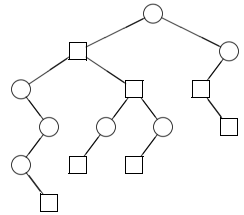

FIG. 1: un exemple d'arbre-dictionnaire

1.1.2 Taille

Démontrer que le cardinal d'un dictionnaire $\mathcal{E}$ est égal au nombre de `Feuille` et `NœudFeuille` figurant dans l'arbre-dictionnaire qui lui est associé.
Écrire une fonction `taille : arbre -> int` qui renvoie le nombre de mots figurant dans le dictionnaire que représente l'arbre argument. Par exemple, dans le cas de l'arbre de la figure 1, cette fonction doit renvoyer l'entier `7`. Bien sûr, on a aussi : $\text{Card}(\{\varepsilon\}) = 1$.

1.1.3 Dénombrement

Soit un arbre-dictionnaire de profondeur p, et $\mathcal{E}$ l'ensemble des mots qu'il représente. Donner un encadrement par des fonctions simples de p du cardinal n de $\mathcal{E}$.

1.2 La fonction μ

Écrire une fonction `mu : arbre -> mot list` qui renvoie la liste de tous les mots figurant dans l'arbre-dictionnaire.
On pourra écrire une fonction `préfixe : Char -> mot list -> mot list` qui ajoute un caractère en tête de chaque mot d'une liste.

1.3 Recherche d'un mot

Écrire une fonction `cherche : mot -> arbre -> bool` qui teste l'appartenance d'un mot à un arbre-dictionnaire.

1.4 Insertion d'un mot

1.4.1 Fonctions auxiliaires

Écrire la petite fonction `nfon : arbre -> arbre` (pour NœudFeuilleOfNœud) qui transforme un `Nœud` en `NœudFeuille` et renvoie tel quel un arbre qui n'est pas un `Nœud`.

Écrire la fonction `arbre1 : mot -> arbre` telle que `arbre1 m` est l'arbre-dictionnaire associé au dictionnaire $\mathcal{E} = \{\texttt{m}\}$.

Écrire la fonction `arbre2 : mot -> mot -> arbre` telle que `arbre2 m1 m2` est l'arbre-dictionnaire associé au dictionnaire $\mathcal{E} = \{\texttt{m1}, \texttt{m2}\}$.

1.4.2 La fonction d'insertion

Écrire la fonction `insère : mot -> arbre -> arbre` chargée de l'insertion d'un nouveau mot dans un arbre-dictionnaire.

1.5 Suppression d'un mot

Écrire la fonction `supprime : mot -> arbre -> arbre` chargée de la suppression d'un mot d'un arbre-dictionnaire (si le mot ne figure pas, on renvoie l'arbre initial inchangé).

2 Arbres et files binomiaux

2.1 Arbres binomiaux

On définit une famille d'arbres, appelés arbres binomiaux par récurrence :

— un arbre binomial de degré 0 noté B_0 comporte juste une racine, sans fils ;

— un arbre binomial de degré k noté B_k comporte une racine possédant k fils dont le premier est un arbre binomial de degré $k-1$, le second un arbre binomial de degré $k-2$,..., le dernier un arbre binomial de degré 0.

2.1.1

Dessiner un arbre binomial de degré 3.

2.1.2

Donner une autre définition récursive des arbres binomiaux, en définissant B_k à partir de deux arbres B_{k-1}.

2.1.3

Démontrer les propriétés suivantes pour un arbre binomial de degré k :

1. Le nombre de nœuds est 2^k.
2. La profondeur d'une feuille est au plus k.
3. Le nombre de sommets à la profondeur p est $\binom{k}{p} = C_k^p$.
4. Tout nœud a au plus k fils.

2.2 Arbre binomial croissant

Un arbre binomial croissant est un arbre binomial où des valeurs sont affectées aux sommets en tant qu'*étiquettes* de façon que chaque sommet possède une valeur supérieure ou égale à celle de son père. Les étiquettes de ces sommets seront appelées les éléments de l'arbre binomial et on parlera pour raison de commodité de la liste des éléments de l'arbre.
On implémente les arbres binomiaux en Caml de la manière suivante, les valeurs affectées aux nœuds étant des entiers :

```
type abinom = Nœud of int * abinom list;;
```

où dans `Nœud(n,l)`, `n` est l'entier affecté à la racine, et `l` est la liste des fils de la racine (qui sont eux aussi des arbres binomiaux).

2.2.1

Écrire une fonction Caml `liste_elts_arbre : abinom -> int list` qui à un arbre binomial associe la liste des entiers qui lui sont affectés.

2.2.2

Écrire une fonction Caml `fusion : abinom -> abinom -> abinom` qui, à deux arbres binomiaux croissants de même degré k associe un arbre binomial croissant de degré $k+1$ dont la liste des éléments est (à l'ordre près) la concaténation des listes des éléments des deux arbres.

2.3 File binomiale

Une file binomiale est une liste d'arbres binomiaux de tailles strictement croissantes. La taille d'une file binomiale est la somme des tailles des arbres binomiaux qui la composent. La liste des éléments d'une file binomiale est la concaténation des listes des éléments des arbres binomiaux qui la composent.

2.3.1

Démontrer que, pour tout entier n, il existe une unique file binomiale de taille n.

2.3.2

On dit qu'une file binomiale est croissante si elle est constituée par des arbres binomiaux croissants (c'est-à-dire que chaque arbre binomial constituant la file est croissant ; il n'y a pas de comparaison de ces arbres entre eux). Une file binomiale sera représentée en Caml par un objet de type `abinom list`.
Écrire une fonction Caml `insère : abinom -> abinom list -> abinom list` qui insère un arbre binomial croissant dans une file binomiale croissante (la liste des éléments de la nouvelle file étant, à l'ordre près, la concaténation des deux listes des éléments de l'arbre et de la file).
Écrire ensuite une fonction `fusion_file : abinom list -> abinom list -> abinom list` qui fusionne deux listes binomiales croissantes.

2.3.3

Utiliser la fonction `insère` pour écrire une fonction `insère_int : abinom list -> int -> abinom list` qui à une file binomiale croissante (dont la liste des éléments est ℓ) et à un entier n associe une file binomiale croissante dont la liste des éléments est la liste ℓ à laquelle est ajouté l'entier n.

2.3.4

Utiliser la fonction `insère` pour écrire une fonction `enlève_min : abinom list -> (int * abinom list)` qui enlève et renvoie l'entier minimum d'une file binomiale croissante f (on pourra soit décrire l'algorithme soit écrire une fonction Caml).

2.3.5

Utiliser les deux fonctions précédentes pour écrire une fonction `tri_binomial : int list -> int list` qui trie une liste d'entiers (on pourra soit décrire l'algorithme soit écrire une fonction Caml).