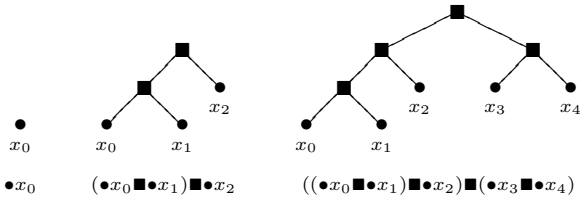
▶ **MPSI – Option Informatique**
Année 2002, Sixième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Arbres binaires et codage de Huffman

1 Arbres binaires

Soit E un ensemble non vide. On définit la notion d'arbre binaire étiqueté (aux feuilles) par E de la manière suivante :
- Si x est un élément de E alors $a = \bullet x$ est un arbre binaire. (x est alors l'unique feuille de a.)
- Si a_0 et a_1 sont deux arbres binaires alors $a = a_0 \blacksquare a_1$ est un arbre binaire. (a_0 est le *sous-arbre gauche* de a et a_1 le *sous-arbre droit*. Les feuilles de a sont exactement celles de ces deux sous-arbres.)

Il est usuel de donner une représentation graphique des arbres sous forme arborescente. Dans celles-ci, nous représenterons les *nœuds internes* par ■ et les feuilles par •. La *racine* d'un arbre est le nœud situé au sommet de ces constructions.



Nous représenterons en *Caml* un arbre binaire étiqueté par des valeurs de type `'a` grâce au type somme récursif `'a tree` défini de la déclaration :

```
type 'a tree =
    Leaf of 'a
  | Node of 'a tree * 'a tree
;;
```

Cette définition de type correspond à la définition formelle de la notion d'arbre que nous avons donnée précédemment. `Leaf x` représente en *Caml* l'arbre à une feuille $\bullet x$. De même, si `a0` et `a1` représentent les arbres a_0 et a_1, `Node (a0, a1)` représente l'arbre $a_0 \blacksquare a_1$.

▶ **Question 1** *Donnez la représentation en Caml des arbres donnés en exemple ci-dessus.*

Les fonctions manipulant de tels arbres sont souvent écrites en utilisant des filtrages et la récursivité (comme pour les listes). Par exemple, la fonction suivante calcule le nombre de feuilles d'un arbre :

```
let rec leaves = function
    Leaf x -> 1
  | Node (a0, a1) -> leaves a0 + leaves a1
;;
```

▶ **Question 2** *Écrivez ainsi une fonction* `nodes` *qui calcule le nombre de nœuds intérieurs d'un arbre.*

`value nodes` : $'a\ tree \to int$

On appelle *hauteur* d'un arbre la longueur maximale d'un chemin *direct* menant de sa racine à une de ses feuilles. Par exemple, les trois arbres donnés en exemple ci-dessus ont respectivement pour hauteur 0, 2 et 3.

▶ **Question 3** *Écrivez une fonction* `height` *qui calcule la hauteur d'un arbre.*

`value height` : $'a\ tree \to int$

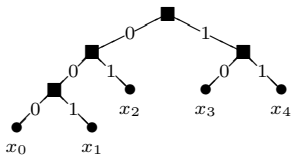
On suppose dans la question suivante que l'ensemble des étiquettes E est ordonné.

▶ **Question 4** *Écrivez une fonction* `max_tree` *qui retourne la plus grande étiquette présente dans un arbre.*

`value max_tree` : $'a\ tree \to' a$

2 Mots binaires

Un *mot binaire* w est une liste finie de booléens (`false` et `true` en *Caml*, notés 0 et 1 dans cet énoncé). Par exemple, le mot binaire 0010 sera représenté en *Caml* par la liste `[ false ; false ; true ; false ]`. Nous considérerons également le mot vide, noté ϵ et représenté par la liste vide.
Étant donné un arbre binaire a, un mot binaire w permet de désigner un sous-arbre de a. Celui-ci est obtenu en parcourant a depuis la racine tout en lisant w bit par bit : à la lecture d'un 0, on descend vers le fils gauche du nœud courant et à la lecture d'un 1 vers le fils droit. Considérons à nouveau l'arbre :



Dans cet exemple, le mot 00 désigne le sous arbre $(\bullet x_0 \blacksquare \bullet x_1)$ alors que 001 désigne $\bullet x_1$. Cependant, les mots 010, 101 ou 0010 ne correspondent à aucun sous-arbre.

▶ **Question 5** *Écrivez une fonction* sub_tree *qui prend pour arguments un arbre a et un mot binaire w. Cette fonction retournera le sous-arbre de a désigné par w. Si le mot binaire w ne désigne pas un sous-arbre de a, vous lèverez une exception.*

value sub_tree : *'a tree → bool list → 'a tree*

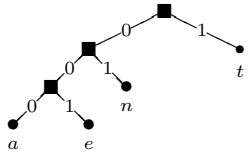
▶ **Question 6** *Écrivez maintenant une fonction* read *qui prend pour arguments un arbre a et un mot binaire w. Cette fonction parcourera l'arbre a en lisant le mot w jusqu'à arriver sur une feuille. La fonction retournera alors l'étiquette de la feuille atteinte et la partie du mot w qui n'a pas été lue. Si le mot w ne permet pas d'atteindre une feuille, votre fonction lèvera une exception.*

value read : *'a tree → bool list → ('a × bool list)*

3 Code de Huffman

3.1 Décodage

Pour représenter les lettres de l'alphabet, les chiffres, les symboles de ponctuation, et d'une manière générale tous les *caractères* qui apparaissent dans un fichier informatique, on associe à chacun d'entre-eux un mot binaire d'une longueur donnée. Dans une représentation habituelle, la longueur du mot binaire est la même pour tous les caractères, un octet par exemple.
Le principe du codage de Huffman est d'associer à chaque symbole du texte à encoder un code binaire qui est d'autant plus court que le caractère correspondant a un nombre d'occurences élevé dans le texte à encoder. Un code de Huffman consiste en un arbre binaire dont les feuilles sont étiquetées par des caractères. Le mot binaire associé à chaque caractère c est celui qui mène de la racine de l'arbre à la feuille étiquetée par c selon la définition donné à la section 2.



On représente alors une chaîne de caractères par la concaténation des mots binaires correspondant à chacun de ses caractères. Par exemple, dans le code précédent, la chaîne *tentant* est représentée par 1001011000011 $(1 \cdot 001 \cdot 01 \cdot 1 \cdot 000 \cdot 01 \cdot 1)$.

▶ **Question 7** *Écrivez une fonction* decode *qui prend pour argument un code de Huffman et un mot binaire. La fonction retournera la chaîne de caractères représentées par le mot binaire dans le code.*

value decode : *char tree → bool list → string*

3.2 Construction du code

Nous nous intéressons maintenant à la construction d'un code de Huffman permettant de représenter un texte d'une manière la plus concise possible. La difficulté réside dans le fait que, pour obtenir un codage efficace, il faut choisir les codes des différents caractères en tenant compte de leurs fréquences respectives. La méthode que nous proposons nécessite de calculer dans un premier temps le nombre d'occurences n_c de chaque caractère c présent dans le texte, afin de former la liste m des couples $(n_c, \bullet c)$. L'algorithme consiste alors à itérer le processus suivant sur la liste m jusqu'à ce que celle-ci soit réduite à un élément :
– Retirer de la liste m les deux couples (n_1, a_1) et (n_2, a_2) tels que n_1 et n_2 soient minimaux,
– Ajouter à la liste m le couple $(n_1 + n_2, a_1 \blacksquare a_2)$.
L'algorithme se termine lorsque la liste m est réduite à un coupe (n, a) : a est alors l'arbre du code de Huffman recherché. Pour obtenir un implémentation raisonnablement efficace, on maintiendra la liste m triée dans l'ordre des n_c croissants. Ainsi, les éléments minimaux apparaîtront toujours en tête.

▶ **Question 8** *Quelle est la liste m obtenue si le texte considéré est le mot* tentant *? Quel est l'arbre construit par notre algorithme ?*

▶ **Question 9** *Écrivez une fonction* insert *prenant pour argument un élément* x *et une liste* q *supposée triée. La fonction retournera la liste obtenue à partir de* q *en ajoutant* x *de manière à maintenir le tri.*

value insert : *'a → 'a list → 'a list*

▶ **Question 10** *Écrivez une fonction* merge *prenant pour argument une liste de couples de la forme $(n_c, \bullet c)$ supposée triée. Cette fonction réduira la liste comme décrit ci-dessus afin d'obtenir le code de Huffman correspondant.*

value merge : *(int × char tree) list → char tree*

▶ **Question 11** *Déduisez-en une fonction* huffman *qui calcule l'arbre de Huffman correspondant à liste de couples (n_c, c) indiquant le nombre d'occurrences de chaque caractère dans un texte.*

value huffman : *(int × char) list → char tree*

3.3 Codage

▶ **Question 12** *Écrivez une fonction* extract *qui prend pour argument l'arbre d'un code de Huffman. Cette fonction calculera la liste des couples (c, w_c) où w_c est le mot binaire représentant le caractère c dans le code.*

value extract : *char tree → (char, bool list) list*

▶ **Question 13** *Déduisez-en une fonction qui prend pour argument un code de Huffman ainsi qu'une chaîne de caractères et qui retourne le mot binaire représentant la chaîne de caractères dans le code de Huffman.*

value code : *char tree → string → bool list*

▶ **Question 14** *Écrivez enfin une fonction* occurrences *qui, étant donnée une chaîne de caractères, calcule la liste attendue par la fonction* huffman *pour construire le code.*

▶ **MPSI – Option Informatique**
Année 2002, Sixième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Arbres binaires et codage de Huffman

Un corrigé

▶ Question 1

```
Leaf "x0"

Node (Node (Leaf "x0", Leaf "x1"),
      Leaf "x2")

Node (Node (Node (Leaf "x0", Leaf "x1"),
            Leaf "x2"),
      Node (Leaf "x3", Leaf "x4"))
```

▶ Question 2

```
let rec nodes = function
    Leaf x -> 0
  | Node (a0, a1) ->
      1 + nodes a0 + nodes a1
;;
```

▶ Question 3

```
let rec height = function
    Leaf _ -> 0
  | Node (a0, a1) ->
      1 + max (height a0) (height a1)
;;
```

▶ Question 4

```
let rec max_tree = function
    Leaf x -> x
  | Node (a0, a1) ->
      max (max_tree a0) (max_tree a1)
;;
```

▶ Question 5

```
let rec sub_tree a w =
  match a, w with
    _, [] -> a
  | Leaf _, _ :: _ ->
    invalid_arg "path"
  | Node (a0, a1), t :: w' ->
    sub_tree (if t then a1 else a0) w'
;;
```

▶ Question 6

```
let rec read a w =
  match a, w with
    Leaf x, _ -> (x, w)
  | Node _, [] ->
      invalid_arg "read"
  | Node (a0, a1), t :: w' ->
      read (if t then a1 else a0) w'
;;
```

▶ Question 7

```
let rec decode a = function
    [] -> ""
  | w ->
    let (c, w') = read a w in
    (string_of_char c) ^ (decode a w')
;;
```

▶ Question 9

```
let rec insert x = function
    [] -> [x]
  | hd :: tl ->
      if x < hd then x :: hd :: tl
      else hd :: (insert x tl)
;;
```

▶ Question 10

```
let rec merge = function
    [] -> invalid_arg "merge"
  | [n, a] -> a
  | (n1, a1) :: (n2, a2) :: tl ->
    merge (insert (n1 + n2, Node (a1, a2)) tl)
;;
```

▶ Question 11

```
let rec huffman m =
  let m' = sort__sort (prefix <=) m in
  merge (map (function n, c -> n, Leaf c) m')
;;
```

▶ Question 12

```
let rec add b = function
    [] -> []
  | (c, w) :: tl ->
      (c, b :: w) :: (add b tl)
;;

let rec extract = function
    Leaf c -> [(c, [])]
  | Node (a0, a1) ->
      (add false (extract a0))
      @ (add true (extract a1))
;;
```

▶ Question 13

```
let code a s =
  let list = extract a in
  let res = ref [] in
  for i = string_length s - 1 downto 0 do
    res := (assoc s.[i] list ) @ !res
  done;
  !res
;;
```

▶ Question 14

```
let occurrences s =

  let t = make_vect 256 0 in
  for i = 0 to string_length s - 1 do
    let j = int_of_char s.[i] in
    t.(j) <- t.(j) + 1
  done;

  let res = ref [] in
  for j = 255 downto 0 do
    if t.(j) > 0 then
      res := (char_of_int j, t.(j)) :: !res
  done;

  !res
;;
```