Correction du rattrapage de programmation fonctionnelle en Objective Caml

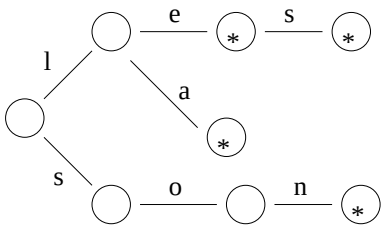
Christian Rinderknecht

19 Juin 2003

Les exercices peuvent être traités de façon indépendante. **Les documents et les calculatrices ne sont pas autorisés.**

1 Lexiques

Un arbre peut représenter une collection de mots de sorte qu'il soit très efficace de vérifier si une séquence donnée de caractères est un mot valide ou pas. Dans ce type d'arbre, appelé un trie*, les arcs ont une lettre associée, chaque nœud possède une indication si la lettre de l'arc entrant est une fin de mot et la suite des mots partageant le même début. La figure suivante montre le* trie *des mots « le », « les », « la » et « son », l'astérisque marquant la fin d'un mot :*



On utilisera le type Caml suivant pour implanter les tries :

```
type trie =
  { mot_complet : string option; suite : (char * trie) list }
```

Le type prédéfini `type 'a option = None | Some of 'a` *sert à représenter une valeur éventuellement absente. Chaque nœud d'un* trie *contient les informations suivantes :*

- *Si le nœud marque la fin d'un mot* `s` *(ce qui correspond aux nœuds étoilés de la figure), alors le champ* `mot_complet` *contient* `Some s`*, sinon ce champ contient* `None`*.*
- *Le champ* `suite` *contient une liste qui associe les caractères aux nœuds.*

A.1 *Écrire la valeur Caml de type* `trie` *correspondant à la figure ci-dessus.*

```
{mot_complet=None;
 suite=[('l',{mot_complet=None;
              suite=[('e',{mot_complet=Some "le";
```

```
                        suite=[('s',{mot_complet=Some "les";
                                     suite=[]})]});
                ('a',{mot_complet=Some "la";
                      suite=[]})]});
        ('s',{mot_complet=None;
              suite=[('o', {mot_complet=None;
                            suite=[('n', {mot_complet=Some "son";
                                          suite=[]})]})]})]}
```

A.2 *Écrire une fonction* `compte_mots` *qui compte le nombre de mots dans un* trie*.*

```
let rec compte_mots trie =
  (match trie.mot_complet with Some _ -> 1 | None -> 0)
+ itérateur 0 trie.suite

and itérateur compte_courant = function
  (_, trie)::reste ->
    itérateur (compte_courant + compte_mots trie) reste
| [] -> compte_courant
```

A.3 *Écrire une fonction* `select` *qui prend en argument une lettre et un* trie*, et renvoie le* trie *correspondant aux mots commençant par cette lettre. Si ce* trie *n'existe pas (parce qu'aucun mot ne commence par cette lettre), la fonction devra lancer une exception* `Absent`*, que l'on définira. On utilisera la fonction prédéfinie* `List.assoc` *:* $\forall\alpha,\beta.\alpha \to (\alpha\times\beta)$ *list* $\to$ β *pour effectuer la recherche dans la liste des sous-arbres* `suite`*.*

```
exception Absent

let select lettre trie =
  try
    List.assoc lettre trie.suite
  with Not_found -> raise Absent
```

A.4 *Écrire une fonction* `appartient` *qui vérifie si une chaîne de caractères est un mot dans un* trie *donné. La fonction devra prendre un argument supplémentaire* `i`*, qui représente la position dans le mot associée au nœud courant. Le* i^e *caractère d'une chaîne* `s` *s'obtient en écrivant* `s.[i]`*, le premier caractère étant numéroté* 0*. La longueur de la chaîne* `s` *s'écrit* `String.length s`*. On emploiera la fonction* `select`*.*

```
let rec appartient chaine index trie =
  if index = String.length chaine
  then match trie.mot_complet with
         Some _ -> true
       | None -> false
  else try
         let fils = select chaine.[index] trie
         in appartient chaine (index+1) fils
       with Absent -> false
```

A.5 *Application : vérificateur orthographique élémentaire. Écrire une fonction* `vérifie` *qui prend une liste de mots, un lexique et retourne la liste des mots du texte n'apparaissant pas dans le lexique.*

```
let rec vérifie lexique = function
  [] -> []
| mot::reste ->
    if   recherche mot 0 lexique
    then vérifie lexique reste
    else mot::(vérifie lexique reste)
```

A.6 *On désire écrire une nouvelle fonction de recherche qui donne une signification spéciale au caractère point (.). Ce caractère se comportera comme un* joker *pouvant remplacer n'importe quelle lettre. Écrire une fonction qui prend en entrée un lexique et un mot pouvant contenir le* joker*, et retourne la liste de tous les mots correspondant.*

À FAIRE.

A.7 *Écrire une fonction qui ajoute un mot à un* trie *(c'est-à-dire qu'elle renvoie un nouveau* trie *contenant en plus ce mot).*

À FAIRE.

2 Tri par fusion (*Merge sort*)

On se propose de programmer un algorithme de tri sur les listes, dit tri par fusion*. L'idée est de commencer par programmer l'opération de fusion, qui prend deux listes déjà triées et en fait une seule liste triée.*

Ensuite, on transforme notre liste à trier, contenant n éléments, en n listes à un élément. Puis on fusionne ces listes deux par deux, ce qui nous donne $n/2$ listes triées à deux éléments, plus éventuellement une liste à un élément (si n est impair). On les fusionne à nouveau deux par deux, etc. jusqu'à ce qu'il ne reste plus qu'une seule liste, qui est alors triée.

Il est intéressant de noter que le type des éléments manipulés n'a pas d'importance, parce que la fonction de comparaison `ordre` *sera polymorphe : elle aura le type $\forall\alpha.\alpha \to \alpha \to$ bool. Cela permet d'utiliser la même fonction* `mergesort` *pour trier des listes d'entiers, de flottants, de listes, etc. sans limitation. De plus, cela permettra de trier par ordre croissant ou décroissant, selon l'implantation de la fonction* `ordre`*.*

B.1 *Écrire une fonction* `singletons` *qui prend en argument une liste* `[x1; ...; xn]` *et renvoie la liste des listes à un élément* `[[x1]; ...; [xn]]`*. On pourra utiliser* `List.map` *: $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha$ list $\to \beta$ list qui calcule la liste des images par une fonction donnée d'une liste d'antécédents donnés.*

```
let singletons l = List.map (fun x -> [x]) l
```

B.2 *Écrire une fonction* `merge` *qui prend en argument la fonction de comparaison* `ordre`*, deux listes triées et les fusionne en une seule liste triée.*

```
let rec merge ordre l1 l2 =
  match (l1, l2) with
```

```
    ([], _) -> l2
  | (_, []) -> l1
  | (x1 :: reste1, x2 :: reste2) ->
      if ordre x1 x2
      then x1 :: (merge ordre reste1 l2)
      else x2 :: (merge ordre l1 reste2)
```

On réalise un filtrage sur la paire `(l1, l2)` (au lieu de faire un filtrage sur `l1` suivi d'un autre sur `l2`) pour plus de concision. Notons que les deux premières branches du `match` se recouvrent dans le cas `([], [])`, ce qui n'est nullement gênant ; la première s'appliquera alors. Remarquons que dans le cas où `x1` est inférieur à `x2`, il serait incorrect de renvoyer `x1 :: x2 :: (merge ordre reste1 reste2)`. En effet, `reste1` peut très bien contenir d'autres éléments compris entre `x1` et `x2`.

B.3 *Écrire une fonction* `merge2à2` *qui prend en argument la fonction de comparaison* `ordre`*, une liste de listes* `[l1; l2; l3; l4; ...]` *et renvoie une liste où les listes voisines ont été fusionnées, i.e.* `[merge ordre l1 l2; merge ordre l3 l4; ...]`*. On prendra garde à traiter correctement le cas où la liste d'entrée est de longueur impaire.*

```
let rec merge2à2 ordre = function
  l1 :: l2 :: reste ->
   (merge ordre l1 l2) :: (merge2à2 ordre reste)
| ll -> ll
```

La première ligne du filtrage s'applique lorsque la liste passée à `merge2à2` contient au moins deux éléments. La seconde ligne s'applique dans tous les autres cas (parce que `ll` est un nom de variable), c'est-à-dire lorsque la liste contient au plus 1 élément.

B.4 *En combinant les fonctions précédentes, écrire une fonction* `mergesort` *qui prend en argument la fonction de comparaison* `ordre`*, une liste et la trie. Pour cela, on crée la liste des listes à un élément, puis on lui applique* `merge2à2` *itérativement jusqu'à obtenir une liste de la forme* `[l]`*. On renvoie alors* `l`*.*

```
let auplus1 = function
  (_ :: _ :: _) -> false
| _ -> true
```

Cette fonction indique si la liste qu'on lui passe contient au plus 1 élément. Le principe est le même que pour `merge2à2`, mais comme on n'a pas besoin d'utiliser les éléments, on utilise `_` au lieu de variables.

```
let rec répète fusion prédicat l =
  if prédicat (l)
  then l
  else répète fusion prédicat (fusion l)

let mergesort ordre l =
  let résultat = répète (merge2à2 ordre) auplus1 (singletons l)
in match résultat with
     [] -> []
   | l :: _ -> l
```

On aurait pu aussi écrire (déconseillé aux âmes sensibles) :

```
let compose f g x = f (g (x))

let mergesort ordre =
  compose List.flatten
          (compose (répète (merge2à2 ordre) auplus1) singletons)
```

Ici, l'application partielle `(merge2à2 ordre)` donne une fonction de type $\forall\alpha.\alpha\,\mathsf{list}\,\mathsf{list} \to \alpha\,\mathsf{list}\,\mathsf{list}$, qui constitue une étape de fusion. On utilise ensuite la fonction `répète` pour effectuer cette étape autant de fois que nécessaire. Plus précisément, `répète` attend trois arguments :

— l'action à répéter, ici `(merge2à2 ordre)` ;
— la condition d'arrêt, ici le prédicat `auplus1`, ce qui signifie que l'on s'arrêtera lorsque la liste de listes ne contiendra plus qu'un argument ;
— le point de départ, à savoir `(singletons l)`, la liste des listes à un élément.

Lorsque `répète` s'arrête, le résultat qu'elle renvoie vérifie nécessairement le prédicat `auplus1` ; il s'agit d'une liste de 0 ou 1 listes triées. On utilise un dernier `match` pour traiter ces deux cas. Dans la deuxième ligne, on sait que la liste a exactement 1 élément ; le motif `_` filtrera en fait toujours une liste vide.

B.5 *Exprimer le nombre maximal de comparaisons nécessité par chacune des fonctions ci-dessus, en fonction de la taille de son argument. En déduire que le tri d'une liste de taille n demande un temps $O(n \cdot \log_2 n)$.*

Nous pouvons maintenant compter le nombre d'opérations effectuées par chacune des fonctions ci-dessus. Lorsque la fonction `merge` se rappelle récursivement, elle a effectué une comparaison, et les listes passées à l'appel récursif contiennent (au total) un élément de moins. Par ailleurs, `merge` s'arrête lorsque l'une des deux listes est vide. Par conséquent, `merge ordre l1 l2` réalise au plus $n_1 + n_2$ comparaisons, où n_i est la taille de `li`.

Ceci étant établi, on vérifie que `merge2à2 ordre [l1; l2; ...]` effectue au plus $n_1 + n_2 + \ldots$ comparaisons. C'est-à-dire au plus n comparaisons, si n est la taille de la liste que nous sommes chargés de trier.

Pour savoir combien `mergesort ordre l` effectue de comparaisons, il faut déterminer combien de fois `merge2à2` est appelée. Or, à chaque appel, elle divise par deux [1] le nombre d'éléments de la liste de listes. Si celle-ci est au départ de taille n, le nombre d'étapes effectuées est donc au plus $\log_2 n$.

Des deux paragraphes précédents, on déduit que le temps nécessaire à `mergesort` pour trier une liste de taille n est au pire de l'ordre de $n \cdot \log_2 n$, ce que l'on écrit $O(n \cdot \log_2 n)$.

3 Tri rapide (*Quicksort*)

C.1 *Écrire une fonction* `partage` *qui prend en argument un prédicat* `p` *et une liste* `l` *et renvoie un couple formé de la liste des éléments de* `l` *qui*

1. En fait, si `ll` est de taille n, alors `merge2à2 ordre ll` est de taille $n/2 + 1$ au plus. Ceci ne change pas le principe de la preuve.

vérifient `p` *et de la liste de ceux qui ne le vérifient pas.*

```
let rec partage p = function
  [] -> ([], [])
| x :: l ->
    let (oui, non) = partage p l
    in if (p x) then (x :: oui, non)
       else (oui, x :: non)
```

C.2 *On choisit un élément* `p` *de la liste, appelé* pivot. *On sépare ensuite la liste en trois parties : le pivot* `p`*, la liste* `l1` *des éléments plus petits que* `p`*, et la liste* `l2` *des éléments plus grands que, ou égaux à* `p`*. (On peut utiliser la fonction* `partage` *écrite précédemment.) La liste triée est alors égale à la concaténation de* `l1` *triée,* `p` *et* `l2` *triée.*

```
let rec quicksort = function
  [] -> []
| pivot::reste ->
    let (petits,grands) = partage (fun x -> x<pivot) reste
    in (quicksort petits) @ (pivot::(quicksort grands))
```