TP 2 - Automates et mots

Jean-Baptiste Rouquier

11 octobre 2005

1 Remarques sur le TP précédent

- Réfléchissez toujours un peu avant de lire les indications.
- N'oubliez pas de tester *toutes* les fonctions que vous écrivez. Même sur un exemple simple.
- N'écrivez pas `[(f x)]@(foo)` mais `f x :: foo`.
- De même évitez les parenthèses superflues. Jamais de parenthèses quand l'argument d'une fonction est un seul token : pas `f(x)` mais `f x`. Bien sûr on conserve les parenthèses dans `f (x+1)`.
  L'application de fonction est prioritaire sur tout sauf
  - l'accès aux tableaux et string (`f vect.(i)`),
  - les opérateurs préfixes (`f !x`).

  Ainsi il n'y a pas besoin de parenthèses dans
  ```
  let rec map f = function
    | [] -> []
    | t :: q -> f t :: map f q
  ```
  (ni autour de `f t` ni autour de `map f q`). En revanche les suivantes sont nécessaires (on suppose que la liste donnée en argument est triée) :
  ```
  let rec dedoublonne = function
    | [] -> []
    | [a] -> [a]
    | a::b::q ->
        if a = b
        then dedoublonne (b :: q)
        else a :: dedoublonne (b :: q)
  ```
- Ne fuyez pas l'impératif. Certes il faut plus pratiquer le fonctionnel car il est moins intuitif. Mais on lit dans les rapports de concours que le correcteur n'aime pas une solution fonctionnelle deux fois plus longue qu'une solution impérative naturelle. D'ailleurs, allez lire les rapports de concours. C'est un énorme avantage de savoir ce que le correcteur attend.
- N'oubliez pas de lire les corrigés sur ma page web. Version papier sur demande. J'y utilise des tournures variées, c'est normal que vous ne les connaissiez pas toutes et que vous me demandiez des explications. N'hésitez pas à me demander le corrigé de la plus longue sous-suite commune si vous l'avez terminé après la séance.
- Appelez-moi si vous avez fait toutes les questions avant la fin...

2 Automates finis déterministes

Remarque. On rappelle que les caractères s'écrivent entre accents graves et non entre apostrophes. Si foo est de type string, on accède à sa n^{e} lettre par `foo.[n]`.

On définit

```
type automate = {
  initial: int;
  finaux: int list;
  arcs: (char * int) list vect}
```

Question 2.1. Faire mentalement la bijection entre la définition mathématique des automates finis déterministes et le type précédent. En cas de panne, me demander un exemple au tableau.

Remarque. Pour gagner en efficacité, on pourrait utiliser des tableaux pour stocker les arcs (`arcs: int vect vect`) et un arbre binaire de recherche (c'est au programme de spé) pour stocker les états finaux.

Aller voir les fonctions `mem` et `assoc` dans la doc. Voici un exemple d'utilisation d'exception (commenté au tableau) :

```
exception Zero;;
let produit_liste liste =
  try
    it_list
      (fun accu elt ->
         if elt=0
         then raise Zero
         else accu * elt)
      1
      liste
  with
    | Zero -> 0
    | Not_found -> failwith
       "Aucun fonction ne peut soulever Not_found dans ce code";;
```

et un autre (`assoc` existe déjà, vous n'avez pas besoin de la redéfinir) :

```
exception Found of int;;
let assoc clef definitions =
  try
    do_list
      (fun (key,def) ->
         if key = clef
         then raise (Found def)) definitions;
    raise Not_found
  with Found def -> def;;
```

Question 2.2. Écrire `accepte: automate -> string -> bool`.

Question 2.3. Dessiner sur papier puis coder (c'est-à-dire créer une valeur de type `automate`) l'automate reconnaissant l'ensemble des mots bien parenthésés *ayant au plus trois niveaux d'imbrication* (`"(()) () ()"` a deux niveaux d'imbrication tandis que `"(() (()) ())"` en a trois). L'alphabet est donc $\{(,)\}$.

Question 2.4. Écrire une fonction `int -> automate` créant l'automate reconnaissant l'ensemble des mots bien parenthésés ayant au plus n niveaux d'imbrication.

> Testez les fonctions écrites jusqu'ici si ce n'est déjà fait. C'est peut-être la dernière fois que je le dis.

On redéfinit le type automate pour qu'au lieu d'obtenir un booléen on obtienne un entier. Au lieu d'un ensemble d'états finaux, l'automate a désormais une fonction `sortie`. La réponse de l'automate est alors donnée par `sortie` appliquée au dernier état du calcul (avant on testait si le dernier état du calcul était final).

```
type automate = {
  initial: int;
  sortie: int -> int;
  arcs: (char * int) list vect}
```

Question 2.5. Écrire `calcul: automate -> string -> int`. On supposera que le calcul est toujours acceptant (sinon on pourrait utiliser le type `calcul: automate -> string -> int option`).

Le mot de Thue-Morse est un mot infini (ou une suite infinie de caractères) sur l'alphabet $\{a, b\}$, qui a la propriété de ne pas contenir de facteur de la forme uuu (ou $u \in \{a, b\}^+$) : il est sans cube. Il est automatique, c'est-à-dire qu'il existe un automate qui sur l'entrée n (écrit en base 2) répond le n^{e} caractère du mot.

Question 2.6. Le n^{e} caractère du mot est le nombre de 1 de l'écriture en base 2 de n, modulo 2. Coder un automate réalisant ce calcul.

Question 2.7. Écrire `begin_thue_morse: int -> string` qui calcule les n premiers caractères du mot de Thue Morse. On pourra utiliser la fonction suivante :

```
(* prévu pour une architecture 32 bits *)
let base_2 n =
  let result = create_string 31 in
  for i=0 to 30 do
    result.[30-i] <-
      (match n land (1 lsl i) with
        | 0 -> '0'
        | _ -> '1')
  done;
  result;;
```

Vérifier que `begin_thue_morse 32` renvoie `"01101001100101101001011001101001"`.

3 Mots

Question 3.1. Écrire la fonction `shuffle: string -> string -> unit` telle que `shuflle u v` imprime tous les mots contenant les lettres de `u` et les lettres

de `v` intercalées. `u` et `v` sont donc des sous-mots de chacun des mots imprimés. Le nom vient de la façon de mélanger un paquet de cartes à l'américaine ou à queue d'aronde (me demander une démo).

Formellement, le shuffle de $u_1 \dots u_n$ et $v_1 \dots v_m$ est

$$\{w_1 \dots w_{m+n} \mid \exists I \subseteq [\![1, m+n]\!] \quad (w_i)_{i \in I} = u \text{ et } (w_i)_{i \notin I} = v\}$$

(on peut colorier les lettres de w en rouge et bleu de façon à ce que les lettres rouges forment u et les bleues forment v).

Indication : on pourra écrire une fonction récursive auxiliaire (éventuellement intégrée à `shuffle`) prenant un préfixe en argument. Elle imprime les mots du shuffle de « la fin de u et v » en imprimant le préfixe devant chacun.

Question 3.2 (théorique, bonus).
Combien y a-t-il d'anagrammes de « Mississipi » ?

Question 3.3 (difficile). Écrire `anagrammes: string -> unit` qui imprime l'ensemble des anagrammes d'un mot donné en argument. On fera attention à n'imprimer chaque anagramme qu'une fois.

Indication : cette question est indépendante de la précédente. On peut écrire une fonction récursive `melanges: string -> (char * int) vect -> unit` telle que `melanges prefixe [|'a',2; 'b',3|]` imprime tous les mots contenant deux a et trois b, en imprimant `prefixe` avant chaque mot. On écrira ensuite quelques fonctions auxiliaires pour transformer le mot dont on veut les anagrammes au format demandé par `melanges`.