%% xyps-pro.tex from $Id: xyps-pro.doc,v 3.11 2011/03/14 20:14:00 krisrose Exp $ %% %% Xy-pic ‘‘Post

TD1 : Listes

1 Structure de données : liste

▷ **Exercice 1**

1. *Écrire la fonction* `deuxieme` *qui renvoie le deuxième élément d'une liste.*
2. *Écrire les fonctions* `n_a_zero` *et* `zero_a_n`*, telles que :*
   `n_a_zero n = [n; n-1;...; 1  ; 0]`
   `zero_a_n n = [0; 1  ;...; n-1; n]`
   Attention : utilisation de `append` *interdite.*
3. *Liste des indices/positions d'un élément e dans une liste l.*

▷ ***Solution***

```
(* CONTRAT
   fonction deuxieme : renvoie le deuxième élément d'une liste
   paramètre liste : 'a list. une liste contenant au moins 2 éléments
   résultat : 'a. le deuxième élément de la liste
*)
let deuxieme liste =
 if liste = [] || (tl liste) = []
 then failwith " deuxieme: pas assez d'éléments"
 else hd (tl liste)
(* TEST
   deuxieme [1; 2; 3] = 2
   deuxieme ['a'; 'b'; 'c'] = 'b'
*)
(* OU BIEN *)
let deuxieme liste =
 match liste with
 | []          -> failwith "deuxieme: liste vide"
 | [p]         -> failwith "deuxieme: un seul élément"
 | x::y::queue -> y

let deuxieme liste =
 match liste with
 | _::x::_ -> x
 | _       -> failwith "deuxieme: pas assez d'éléments"

(* CONTRAT
   fonction n_a_zero : renvoie la liste des entiers de n à 0
   paramètre n : int. entier naturel
   résultat : int list
*)
let rec n_a_zero n =
   if n < 0 then [] else n :: n_a_zero (n-1)
(* TEST
   n_a_zero (-1) = []
   n_a_zero 0 = [0]
   n_a_zero 3 = [3; 2; 1; 0]
```

```
*)

(* CONTRAT
   fonction zero_a_n : renvoie la liste des entiers de 0 à n
   paramètre n : int. entier naturel
   résultat : int list
*)
let rec zero_a_n n =
 if n < 0 then [] else zero_a_n (n - 1) @ [n]
(* TEST
   zero_a_n (-2) = []
   zero_a_n 0 = [0]
   zero_a_n 3 = [0; 1; 2; 3]
*)

(* OU BIEN en version efficace, complexité en Theta(n) *)
let zero_a_n n =
 let rec zero_a_n_term p liste_p_plus_1_a_n =
  if p < 0 then liste_p_plus_1_a_n
           else zero_a_n_term (p-1) (p::liste_p_plus_1_a_n)
 in zero_a_n_term n []
zero_a_n : int -> int list = <fun>
```

Il faut parler de la complexité de `n_a_zero` *et* `zero_a_n`*. Pour des fonctions sur les listes, une mesure raisonnable du temps de calcul est de compter le nombre de fois où un élément de la liste est parcouru par une fonction, i.e. le nombre d'appels de ces fonctions et de la fonction* `@`*. Pour* `n_a_zero`*, c'est élémentaire. Pour* `zero_a_n`*, il y a bien sûr n appels, qui chacun réalisent un appel à* `@`*. Chaque appel à* `@` *est linéaire en terme de longueur de son opérande de gauche, ils sont donc de complexité n, puis $n-1$, puis $n-2$, etc, appels de* `@`*. On obtient $\Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$.*

Enfin, j'ai défini `n_a_zero` *et* `zero_a_n` *pour toutes les valeurs de n, même négatives, par "continuité". La solution où on lève une exception est bien sûr possible, mais il n'y a pas vraiment lieu de le faire, puisqu'on peut définir quelque chose de raisonnable.*

```
(* CONTRAT
   fonction positions : renvoie la liste des positions d'un élément dans une liste
   paramètre element : 'a. l'élément à chercher
   paramètre liste : 'a list. la liste dans laquelle chercher
   résultat : int list. liste des positions (par ordre croissant, commençant à 0)
                        de l'élement dans la liste.
let positions element liste =
 let rec parcours liste indice =
  match liste with
  | []          -> []
  | tete::queue -> if tete = element then indice :: parcours queue (indice+1)
                                     else          parcours queue (indice+1)
  in parcours liste 0
(* TEST
   positions 'a' [] = []
   positions 'a' ['a'; 'a'; 'a'] = [0; 1; 2]
   positions 0 [1; 2; 3] = []
   positions 3 [1; 2; 3; 4] =[2]
*)
```

```
(* OU BIEN directement récursif structurel,
   sans passer par un paramètre auxiliaire de position *)
let rec positions element liste =
 match liste with
 | []          -> []
 | tete::queue -> let positions_dans_queue = (positions element queue) in
                  let positions_decalees   = map (fun pos -> pos+1) positions_dans_queue
                  in if tete = element then      positions_decalees
                                     else 0 :: positions_decalees
```

Si le temps est suffisant, on peut présenter les deux solutions, et les comparer selon :

- *la facilité de construction de la solution : la seconde version est "forcée". En effet, si on utilise la décomposition récursive structurelle de la liste, il faut utiliser les positions dans la queue pour définir les positions dans la liste entière. Donc, il faut décaler les positions, et ajouter un* 0 *ou non en tête. La première solution nécessite l'introduction d'un paramètre auxiliaire "arbitraire" qui ne découle pas directement du problème.*
- *la complexité de la solution : la première solution est nettement meilleure, on ne parcourt qu'une seule fois chaque élément de la liste pour le comparer à l'élément cherché. Complexité en* $\Theta(length(liste))$*. La seconde version, à cause de la nécessité du décalage, oblige à parcourir le premier élément* 1 *fois, le deuxième* 2 *fois, etc. Complexité quadratique en* $\Theta(length(liste)^2)$*.*

Bon, à vrai dire, il n'y a pas lieu de se fâcher, puisqu'en fait la première version peut s'obtenir par un procédé de transformation de code/optimisation "classique" de la seconde version.

▷ **Exercice 2**

Utilisation des itérateurs obligatoire.

1. *Écrire* `map`*, qui applique une fonction donnée à tous les éléments d'une liste, i.e.* *`map f [a; b; c] = [f a; f b; f c]`.*
2. *Écrire* `flatten` *(aplatissement d'une liste de listes).*
3. *Écrire une fonction* ***`fsts`*** *qui prend une liste de couples et renvoie la liste des premiers éléments.*
4. *Écrire une fonction* ***`split`*** *telle que :* `split [(a1,b1);...;(an,bn)]=([a1;...;an],[b1;...;bn])`.
5. *Écrire une fonction qui supprime les doublons d'une liste.*

▷ ***Solution***

```
(* CONTRAT
   fonction map : applique une fonction à tous les éléments d'une liste
   paramètre fonction : 'a -> 'b. la fonction à appliquer
   paramètre liste : 'a list.
   résultat : 'b list. la liste résultant de l'application de la fonction
                       aux éléments de la liste
*)

let map f l = List.fold_right (fun e map_q -> (f e)::map_q) l []

(* TEST
   map (fun x -> x) [1; 2; 3] = [1; 2; 3]
   map (fun x -> x*x) [] = []
   map (fun x -> x*x) [-1; 2; -3; 4] = [1; 4; 9; 16]
*)
```

```
(* CONTRAT
   fonction flatten : aplatit une liste de liste, i.e.
                      flatten [ []; [1; 2]; [3]; [4; 5; 6] ] = [1; 2; 3; 4; 5; 6]
   paramètre liste_listes : 'a list list. une liste de listes
   résultat : 'a list. la liste égale à la concaténation de toutes les listes
*)
let rec flatten liste_listes =
 match liste_listes with
 | []                -> []
 | liste_tete::queue -> append liste_tete (flatten queue)


let flatten ll = List.fold_right (fun e flatten_q -> e@flatten_q) ll []

(* TEST
   flatten [] = []
   flatten [ [] ] = []
   flatten [ [1; 2]; []; [3] ] = [1; 2; 3]
   flatten [ []; [1; 2]; [3]; [4; 5; 6] ] = [1; 2; 3; 4; 5; 6]
*)


let fsts = List.map fst

let split l = List.fold_right (fun (e1,e2) (split_q_1,split_q_2) -> (e1::split_q_1,e2::split_q_2)) l

let remove l = List.fold_right (fun e remove_q -> if List.mem e remove_q then remove_q else e::remove
```

---

2 Modules, application aux files

2.1 Type abstrait

▷ **Support étudiant**

Dans une spécification, on trouve plutôt des déclarations de types que des définitions. On parle alors de **types privés** ou **types abstraits**. L'absence de définition de types interdit à l'utilisateur de manipuler les valeurs “à la main” et n'importe comment, l'obligeant donc à employer les fonctions de manipulation fournies.

Si, en plus des déclarations de types et de fonctions, on trouve des équations qui spécifient la sémantique de ces fonctions (sans faire apparaître une implantation particulière), on parle alors de **types abstraits algébriques**. Bien sûr, ces équations ne sont pas supportées par les langages de programmation.

2.2 Un exemple : les ensembles

2.2.1 Spécification : Les ensembles

▷ **Support étudiant**

```
(***** le type abstrait = pas de definition             *****)
type 'a set ;;

(***** les constructeurs abstraits                       *****)
    (* Fonction vide : construit l'ensemble vide             *)
val vide         : unit -> 'a set;;
    (* Fonction singleton : constuit un ensemble a un element *)
    (* Parametre : a l'element                                *)
    (* Retour : l'ensemble {a}                                *)
val singleton    : 'a -> 'a set;;
    (* Fonction union : realise l'union de deux ensembles    *)
    (* Parametres : ens1 et ens2 deux ensembles              *)
    (* Retour : l'union ensembliste de ens1 et ens2          *)
val union        : 'a set -> 'a set -> 'a set;;

(***** les accesseurs ou requetes ou selecteurs          *****)
    (* Fonction est_vide : teste si un ensemble est vide     *)
val est_vide     : 'a set -> bool;;
    (* Fonction choix : choisi un element de l'ensemble      *)
    (* Precondition : l'ensemble ne doit pas etre vide       *)
val choix        : 'a set -> 'a;;
    (* Fontion appartient :
       teste si un element est dans l'ensemble               *)
val appartient   : 'a -> 'a set -> bool;;

(***** les destructeurs                                  *****)
    (* Fonction enleve : retire un element a l'ensemble      *)
    (* Parametre e : l'element a enlever                     *)
    (* Parametre ens : l'ensemble                            *)
    (* Retour : ens\{e}                                      *)
val enleve       : 'a -> 'a set -> 'a set;;
    (* Fonction intersection : realise l'intersection de *****)
    (* deux ensembles                                        *)
    (* Parametres : ens1 et ens2 deux ensembles              *)
    (* Retour : l'intersection ensembliste de ens1 et ens2   *)
val intersection : 'a set -> 'a set -> 'a set;;
```

Listing 1 – ensemble.mli

Les constructeurs “abstraits” jouent exactement le même rôle que de véritables constructeurs, sauf qu’ils ne montrent rien de l’implantation (ce sont juste des fonctions).

S’il fallait en faire un TAA, on aurait par exemple les propriétés suivantes :

```
est_vide (vide ()) = true
not (appartient x (vide ())) = true
appartient x (singleton y) = (x = y)
appartient x (union ens ens') = appartient x ens || appartient x ens'
...
```

Remarque : on a déjà vu une implantation possible des ensembles par des listes sans doublons. On en verra d’autres (ABR, arbres à gauche, etc).

2.2.2 Une implantation POSSIBLE : Les listes

▷ **Support étudiant**

```
(* definition du type 'a set *)
type 'a set = 'a list ;;
```

```
(* les constructeurs abstraits *)
let vide () = [];;

let singleton e = [e];;

let rec union ens1 ens2 =
  List.fold_right (fun t tq -> if (List.mem t ens2) then tq else t::tq) ens1 ens2 ;;

(* les accesseurs ou requetes ou selecteurs *)
let est_vide ens = (ens=[]);;

let choix ens =
  match ens with
  | [] -> failwith "Ensemble vide!";
  | t::q ->t;;

let appartient = List.mem ;;

(* les destructeurs *)
let rec enleve e ens =
  match ens with
  | [] -> []
  |t::q -> if (e=t) then q else t::(enleve e q);;

let rec intersection ens1 ens2 =
  List.fold_right (fun t tq -> if (appartient t ens2) then t::tq else tq) ens1 [] ;;
```

Listing 2 – ensemble.ml

2.3 Un exemple de TAA : Le module file

▷ Support étudiant

La file est une structure de données linéaire qu'on retrouve partout. On peut enfiler des éléments à une extrémité et les retirer à l'autre extrémité, comme un pipe-line où circulerait un "flot" de valeurs. La file est une structure de type FIFO (*first in, first out*).

▷ Exercice 3

- *Proposer une spécification des opérations du TAA file.*
- *Proposer une implantation fonctionnelle simple à base de listes.*
- *Évaluer la complexité des opérations.*

▷ *Solution*

Signature similaire à celle des piles.

```
(* FICHIER file.mli *)
(* le type abstrait *)
type 'a file

(* les constructeurs abstraits *)
val vide    : unit -> 'a file
val enfiler : 'a -> 'a file -> 'a file

(* les autres fonctions *)
val est_vide : 'a file -> bool
val defiler  : 'a file -> 'a * 'a file
```

Équations du TAA (trop compliquées pour être présentées) :

```
est_vide (enfiler x f)        = false
est_vide (vide ())            = true
defiler (enfiler x (vide ())) = x,
                                vide ()
defiler (enfiler x f)         = fst (defiler f),
                                enfiler x (snd (defiler f))  <=  f <> vide ()
```

Une implantation "naïve" à base de listes est triviale si on choisit d'ajouter en fin de liste :

```
(* FICHIER file.ml *)
(* définition du type file *)
type 'a file = 'a list

(* les constructeurs abstraits *)
let vide () = []
let enfiler x file = file @ [x]

(* les autres fonctions *)
let est_vide file = file = []

let defiler file =
 match file with
 | []   -> failwith "defiler: file vide"
 | t::q -> t, q
```

La complexité des opérations est en $\Theta(1)$, sauf pour `enfiler`, qui est en $\Theta(taille(file))$!

Maintenant, on va proposer une autre implantation efficace (autant qu'avec un tableau et dynamique contrairement au tableau) du TAA file. Mais d'abord, parlons de **complexité amortie**.

2.4 La complexité amortie

▷ **Support étudiant**

On a déjà vu la complexité moyenne et du pire cas, qui s'intéressent seulement à une exécution donnée d'un algorithme. La complexité amortie permet de lisser les temps de calcul des différents appels à un algorithme, supposé inséré dans une application plus grosse. C'est une moyenne temporelle sur toutes ses exécutions. Cette mesure est donc plus fiable quant au temps réel passé dans un algorithme donné. Cela permet de prouver par exemple que quelques exécutions de mauvaise complexité (du pire cas) peuvent être compensées à la longue par d'autres exécutions plus favorables.

2.5 Le TAA file revisité

Idée à faire trouver : la mauvaise complexité linéaire de `defiler` est due au fait qu'on doit parcourir la liste en entier pour manipuler la fin. Il serait idéal de faire que la fin de la liste soit à l'envers, on pourrait ainsi aisément la manipuler, quelque chose comme :

$$t_1 :: t_2 :: \ldots ??? \ldots :: t_{n-1} :: t_n$$

Ainsi, t_n ne serait pas une queue, mais une tête de liste. Le problème est ???. On ne peut pas créer une structure unique de cette nature étrange par typage en OCaml. Mais on peut en créer deux ! ! Une pour le début jusqu'au ???, une autre à l'envers depuis la fin jusqu'au ???.
Ainsi, une file sera spécifiée par une paire de listes OCaml `(debut, rev_fin)` telle que la séquence des éléments dans la file soit `debut@(List.rev rev_fin)`.

```
type 'a file = 'a list * 'a list
```

▷ **Exercice 4**

— *Trouver une implantation efficace du TAA file à l'aide de cette représentation.*

— *Étudier la complexité des opérations.*

▷ ***Solution***

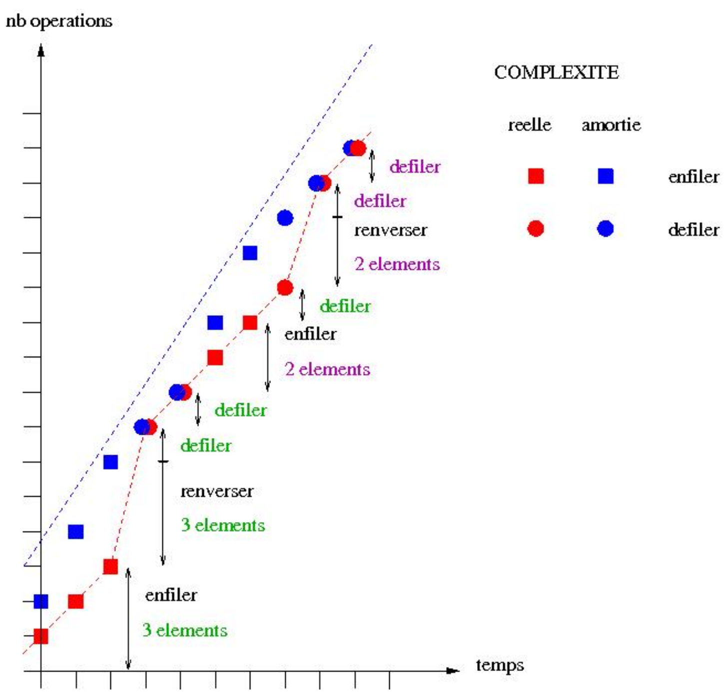
```
let vide () = ([], [])
let est_vide (debut, rev_fin) = debut = [] && rev_fin = []
let enfiler x (debut, rev_fin) = (x::debut, rev_fin)
let defiler (debut, rev_fin) =
 match rev_fin with
 | t::q -> (t, (debut, q))
 | []   -> match List.rev debut with
           | []   -> failwith "defiler: file vide"
           | t::q -> (t, ([], q))
```

`defiler` reste de complexité linéaire dans le pire cas, malgré nos efforts (les autres opérations étant en temps constant). Mais le renversement de liste ne va pas se produire tout le temps et de plus la liste renversée n'a pas toujours une taille égale à la taille de la file ! !

Comment estimer le temps de calcul véritable du `List.rev` *et donc de* `defiler`*, dans une application où de multiples appels aux différentes opérations sont effectués sur une file ?*

A un instant donné, le `List.rev` *ne peut avoir parcouru (et renversé) que les p éléments déjà insérés dans la file depuis le début de l'exécution. Ces p éléments ont été insérés par p appels à* `enfiler`*, qui est en $\Theta(1)$. Donc, le temps global pour* `enfiler` *est proportionnel à p. C'est aussi le temps de* `List.rev` *et donc de* `defiler`*.*

On applique maintenant la **méthode du banquier***. C'est une méthode classique de calcul de complexité amortie, qui peut être rendue rigoureuse, où certaines fonctions peuvent "emprunter" de la complexité à d'autres fonctions, qu'elles vont "rendre" au fur et à mesure de l'exécution. Ici, la complexité de chaque étape de renversement (pas toute la liste, juste un chaînon) est empruntée par* `enfiler`*, qui reste ainsi en $\Theta(2) = \Theta(1)$. Et la complexité de* `defiler` *est donc maintenant en $\Theta(1)$ également, puisque* `List.rev` *ne lui coûte plus rien (le coût étant payé par le remboursement fait par* `enfiler`*) ! ! Ce raisonnement n'est correct que si une fonction ne prête (une partie de) sa complexité qu'à d'autres appels de fonctions déjà exécutés (et non pas à des appels de fonctions dans un hypothétique futur). C'est bien le cas ici, puisque pour retourner la liste, il faut déjà y avoir enfilé les éléments.*



nb operations
COMPLEXITE
reelle
amortie
enfiler
defiler
defiler
defiler
renverser
2 elements
defiler
enfiler
2 elements
defiler
defiler
renverser
3 elements
enfiler
3 elements
temps