▶ **MP & MP* – Option Informatique**
Premier TP Caml
Mardi 19 et 26 septembre 2006

Mise en jambe CamlLight

Quelques informations utiles pour la séance :
- *la page officielle de CamlLight sur le site de l'INRIA : http ://caml.inria.fr/caml-light/index.fr.html ,*
- *la documentation tout aussi officielle : http ://caml.inria.fr/pub/docs/manual-caml-light/ (les parties* The core library *et* The standard library sont les plus intéressantes*),*
- *ne surtout pas hésiter à tester les algorithmes sur papier (notamment les cas extrêmes $i = 0, 1, \ldots$?),*
- *ne pas hésiter non plus à utiliser la fonction* print_int *ou* print_float.
- *s'il y a des questions, il paraît que je suis là pour ça... ;)*

1 Utilisons la syntaxe de Caml

1.1 Fonctions curryfiées ou non

En CamlLight, il y a deux façons possibles de transmettre des données à une fonction :
- en *curryfié*, en donnant plusieurs arguments :
  let ma_fonction x y z = 0;;
- en non-*curryfié*, de donnant un seul couple d'arguments (ou tuple) :
  let ma_fonction (x,y,z) = 0;;
  Ici, (x,y,z) constitue un seul argument (un peu comme un vecteur de taille 3).

La plupart des langages de programmation sont curryfiés mais utilisent une syntaxe semblable à la syntaxe non-curryfiée de Caml.

▶ **Question 1** *Ecrivez une fonction* curryfie *qui transforme une fonction non-curryfiée en fonction curryfiée (à deux arguments). Ecrivez également une fonction* decurryfie *qui fait le contraire.*

```
curryfie : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

decurryfie : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

1.2 Composition de fonctions en Caml

▶ **Question 2** *Ecrivez une fonction* compose *qui renvoie la fonction composée $f \circ g$ des fonctions f et g. Utilisez l'opérateur* #infix *"mafonction" pour obtenir une fonction* o *tel que* (f o g) x *soit égal à $f \circ g(x)$.*

1.3 Quelques opérations sur les listes

▶ **Question 3** *Ecrivez une fonction* iter *qui applique une fonction quelconque* f *à tous les éléments d'une liste. De la même façon, écrivez une fonction* rev_iter *qui applique une fonction tout aussi quelconque* f *aux éléments d'une liste pris dans l'ordre inverse. On se contente d'appliquer* f *aux éléments, sans renvoyer de résultat.*

```
iter : ('a -> 'b) -> 'a list -> unit
rev_iter : ('a -> 'b) -> 'a list -> unit
```

▶ **Question 4** *Déduisez-en une fonction* print_int_list *la plus simple possible.*

```
print_int_list : int list -> unit
```

Il y a quelques fonctions sur les listes qu'il est utile de connaître pour ne pas réinventer la roue systématiquement :

```
value rev : 'a list -> 'a list
```

Cette fonction renvoie la liste inversée (les premiers seront les derniers, et les derniers seront les premiers).

```
value map : ('a -> 'b) -> 'a list -> 'b list
```

map est une fonction qu'on retrouve dans la plupart des langages de programmation et qui applique une fonction f à tous les éléments d'une liste, renvoyant la liste des résultats (dans l'ordre d'origine).

```
value do_list : ('a -> unit) -> 'a list -> unit
```

do_list fait exactement la même chose que notre jolie fonction iter . D'ailleurs, comment peut-on modifier iter pour la forcer à avoir le même prototype que do_list ?

```
value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Si on utilise `it_list f a [b1; ...; bn]`, alors on a `f` qui prend un `'a` et un `'b` et qui renvoie un `'a` et le résultat final est `f (... (f (f a b1) b2) ...) bn`.

```
value list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Si on utilise `list_it f [a1; ...; an] b`, alors on a `f` qui prend un `'a` et un `'b` et qui renvoie un `'b` et le résultat final est `f a1 (f a2 (... (f an b) ...))` . Elle marche un peu comme la fonction précédente, sauf qu'on applique `f` d'abord au dernier élément de la liste et non au premier.

▶ **Question 5** *Utilisez* `list_it` *ou* `it_list` *pour écrire une fonction* `sum_list` *qui calcule la somme des éléments d'une liste.*

```
sum_list : int list -> int
```

▶ **Question 6** *Servez-vous de* `it_list` *pour réécrire la fonction* `rev` *puis une fonction* `rev_map` *(cette dernière fonction va être comme* `map`*, mais elle doit renvoyer les résultats dans l'ordre inverse).*

```
rev : 'a list -> 'a list
rev_map : ('a -> 'b) -> 'a list -> 'b list
```

2 Le tri fusion, avec des listes

Petit rappel :
Comme souvent en informatique, on va progresser lentement, et on va décomposer l'algorithme en petites étapes simples. Le tri fusion opère récursivement : il découpe la liste complète donnée en argument en deux sous-listes l_1 et l_2 de longueurs plus ou moins égales, il trie ces sous-listes avec un appel récursif et il reconstitue la liste complète triée en fusionnant l_1 et l_2.

▶ **Question 7** *Ecrivez une fonction* `half` *qui scinde une liste l en deux sous-listes de tailles égales (à un élément près). La liste ne devra être parcourue qu'une seule fois au total.*

```
half : 'a list -> 'a list * 'a list
```

▶ **Question 8** *Ecrivez une fonction* `fusion` *qui fusionne deux listes triées en une seule liste (toujours aussi bien triée, évidemment !).*

```
fusion : 'a list -> 'a list -> 'a list
```

▶ **Question 9** *Finissez d'écrire la fonction* `tri_fusion` *en utilisant bien sûr les fonctions précédentes.*

```
tri_fusion : 'a list -> 'a list
```

3 Deux tris sur les tableaux (ou : les joies de l'impératif)

Contrairement aux listes, les tableaux ne se plient pas bien aux fonctions récursives, et c'est une excellente occasion de tester deux autres algorithmes de tri très classiques.

3.1 Le tri à bulle

Il s'agit sûrement du tri le plus simple : tant qu'on trouve deux éléments qui ne sont pas rangés dans le bon ordre, on les permute et on recommence à parcourir le tableau.

▶ **Question 10** *Quelle est la complexité en temps (dans le pire cas) de cet algorithme ?*

▶ **Question 11** *Ecrivez une fonction qui échange les éléments d'indice* `i` *et* `j` *dans le tableau* `v`*.*

```
echange : 'a vect -> int -> int -> unit
```

▶ **Question 12** *Ecrivez une fonction* `tri_bulle` *ayant le prototype suivant :*

```
tri_bulle : 'a vect -> 'a vect
```

3.2 Le tri rapide (ou *quicksort*)

Le tri rapide (ou tri de Hoare, du nom de son inventeur) est le plus souvent utilisé pour sa complexité moyenne en $n\log(n)$. Dans cette partie, nous considérons un vecteur `v` de taille `v`. Le tri rapide consiste d'abord à séparer `v` en trois morceaux :

- un pivot `x`, par exemple le premier élément de `v`,
- un sous-tableau `v1` constitué des éléments de `v` inférieurs à `x`,
- un second sous-tableau `v2`, constitué des éléments de `v` supérieurs à `x`.

On trie alors récursivement `v1` et `v2` et on concatène `v1`, `x` et `v2` pour obtenir le tableau `v`, trié.
Maintenant, nous allons couper `v` en deux, en prenant le premier élément de `v` comme pivot.

▶ **Question 13** *Ecrivez une fonction* `decoupe` *qui réorganise le sous-vecteur* `v(a..b)` *de telle sorte que tous les éléments d'indice inférieur à* `c` *soient inférieurs à* `x = v.(a)`*, tous les éléments d'indice supérieur à* `c` *soient supérieurs à* `x` *et enfin que* `v.(c) = x`*.* `decoupe` *devra retourner* `c` *en résultat.*

```
decoupe : 'a vect -> int -> int -> int
```

Indication : on pourra parcourir `v` *à la fois de gauche à droite et de droite à gauche, en échangeant un élément de gauche avec un élément de droite. On placera* `x` *après avoir fait cela.*

▶ **Question 14** *En s'aidant de la fonction précédente, écrivez une fonction qui trie le sous-vecteur* `v(a..b)` *par ordre croissant.*

```
tri_partiel : 'a vect -> int -> int -> unit
```

▶ **Question 15** *Finissez d'écrire la fonction de tri rapide. S'il vous reste du temps, modifiez les fonctions précédentes pour pouvoir utiliser une fonction de comparaison générique.*

```
tri_rapide : 'a vect -> unit
```

▶ **MP & MP* – Option Informatique**
Premier TP Caml
Mardi 19 et 26 septembre 2006

Mise en jambe CamlLight

Un corrigé

▶ Question 1

```
let decurryfie f = fun (x,y) -> f x y;;
let curryfie f = fun x y -> f (x,y);;
```

▶ Question 2

```
let compose = fun f g x -> f (g x);;
let compose f g x =
f (g x);;
let o = fun f g x -> f (g x);;
#infix "o";;
let next = fun x -> x + 1;;
let mul x = 2 * x;;
(next o mul) 3;;
(mul o next) 3;;
```

▶ Question 3

```
let rec iter f list =
match list with
| t::q ->f t; iter f q
| [] -> ();;
let rec rev_iter f list =
match list with
| t::q -> rev_iter f q; f t;
| [] -> ();;
```

▶ Question 4

```
let print_int_list v = iter print_int v;;
```

▶ Question 5

```
let sum_list list =
it_list (fun base elt -> base + elt) 0 list;;
sum_list [1; 2; 3];;
```

▶ Question 6

```
let rev list =
it_list (fun liste elt -> elt::liste) [] list;;
rev [4; 3; 2; 1];;
let rev_map f list =
it_list (fun list elt -> (f elt)::list) [] list;;
rev_map next [1;3;5;7];;
```

▶ Question 7

```
let rec half list =
match list with
| t::q -> let (a,b) = half q in (t::b, a)
| [] -> ([], []);;
half [1; 2; 3; 4; 5; 6; 7; 8];;
```

▶ Question 8

```
let rec fusion l1 l2 =
match (l1, l2) with
| (t1::q1, t2::q2) when (t1 < t2) -> t1::(fusion q1 l2)
| (t1::q1, t2::q2) -> t2::(fusion l1 q2)
| (_, []) -> l1
| ([], _) -> l2
;;
```

▶ Question 9

```
let rec tri_fusion list =
match list with
| t::[] -> list
| t::q -> let (a,b) = half(list) in
fusion (tri_fusion a) (tri_fusion b)
| [] -> [];;
tri_fusion [9; 1; 2; 8; 4; 5];;
```

▶ Question 11

```
let echange v i j = let x = v.(i) in
v.(i) <- v.(j); v.(j) <- x;;
```

▶ Question 12

```
let tri_bulle vect =
let condition = ref true in
let n = vect_length vect in
while !condition do
condition := false;
for i = 0 to (n - 2) do
if(vect.(i) > vect.(i+1)) then
begin
echange vect i (i+1);
condition := true;
end
done;
done;
vect;;
tri_bulle [|4; 2; 8; 10; 2|];;
```

▶ Question 13

```
let decoupe v a b =
let i = ref(a+1) and j = ref b and x = v.(a) in
while !i <= !j do
while(!i <= !j) & (x >= v.(!i)) do i := !i+1 done;
while(!i <= !j) & (x <= v.(!j)) do j := !j−1 done;
if(!i < !j) then begin
echange v !i !j;
i := !i+1;
j := !j−1;
end
else if !i = !j then j := !j−1;
done;
if a <> !j then echange v a !j;
!j
;;
```

▶ Question 14

```
let rec tri_partiel v a b =
let c = decoupe v a b in
if a < c−1 then tri_partiel v a (c−1);
if c+1 < b then tri_partiel v (c+1) b;
```

▶ Question 15

```
let tri_rapide v =
let n = vect_length v in
if (n >= 2) then tri_partiel v 0 (n−1);;
tri_rapide [|4;5;1;3;10;3|];
```