▶ **MP & MP* – Option Informatique**
Premier TP Caml
Mardi 11 et 18 septembre 2007

# 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... ;)*
- *et si besoin est, mon adresse électronique : matthieu.gallet@ens-lyon.fr*

Notions : fonctions, listes, exponentielle rapide, tri par insertion, tri fusion

## 1 Utilisons la syntaxe de Caml

### 1.1 Les différentes syntaxes pour écrire une fonction

En CamlLight, il y a (au moins) 3 méthodes possibles pour écrire une fonction simple.
Considérons l'exemple $\mathbf{f} : x \to 2x$. Nous pouvons écrire :
- **let** f = **fun** x −> 2 ∗ x
- **let** f = **function** x −> 2 ∗ x
- **let** f x = 2 ∗ x

Cependant, il existe des subtilités entre ces 3 écritures.
- **fun** permet d'utiliser un nombre quelconque d'arguments,
- **function** est limitée à un seul argument,
- la dernière méthode ne permet pas d'utiliser directement le « pattern matching ».

▶ **Question 1** *Écrivez avec les 3 méthodes décrites les fonctions mathématiques suivantes :*
- $\boldsymbol{f} : x \to x^2$
- $\boldsymbol{g} : x \to 1$ si $x = 0$, $x\boldsymbol{g}(x-1)$ sinon
- $\boldsymbol{h} : (x, y) \to x + y$

### 1.2 Fonctions curryfiées ?

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 (C, Maple, Java, ...) utilisent plusieurs arguments (donc une syntaxe curryfiée), mais en les séparant avec des virgules et des parenthèses.
Plus concrètement, les deux approches sont équivalentes et vous pouvez les utiliser indifféremment tant que vous respectez la syntaxe des fonctions demandées.

▶ **Question 2** *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
```

### 1.3 Composition de fonctions en Caml

▶ **Question 3** *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* $f \circ g(x)$.

# 2 Fonctions récursives classiques

## 2.1 Fonction exponentielle

Nous cherchons dans cette partie à calculer $x^n$, où $x$ est un entier strictement positif.

▶ **Question 4** *Écrivez en Caml une fonction simple qui calcule* $x^n$

```
exp : int -> int -> int
```

Combien de multiplications sont nécessaires pour mener à bien le calcul ?

▶ **Question 5** *Si ce n'est pas déjà le cas, écrivez une fonction qui calcule* $x^n$ *avec une complexité de* $\log n$ *multiplications. On parle alors d'« exponentielle rapide ».*

```
exp : int -> int -> int
```

## 2.2 Un autre exemple simple : la suite de Fibonacci

## 2.3 Une fonction qui ne sert à rien : la fonction d'Ackermann

# 3 Utilisons les listes

## 3.1 Quelques opérations sur les listes

▶ **Question 6** *Écrivez 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 7** *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 8** *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 9** *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
```

## 3.2 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 10** *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 11** *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 12** *Finissez d'écrire la fonction* tri_fusion *en utilisant bien sûr les fonctions précédentes.*

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

## 3.3 Le tri par insertion, toujours avec des listes

Même si ce tri est plus adapté à une programmation impérative, nous allons l'implémenter avec des listes (notamment pour de basses raisons de TP trop court).

▶ **Question 13** *Comme son nom l'indique, le tri par insertion nécessite de faire des insertions dans une liste triée. Écrivez donc en Caml une fonction* insert *, telle que* insert l t *insère dans* l *l'élément* t*, de façon à ce que* l *reste triée.*

```
insert : 'a list -> 'a -> 'a list = <fun>
```

▶ **Question 14** *Maintenant, nous allons finir le tri par insertion : pour cela, nous allons récupérer un par un les éléments de la liste d'origine, et les insérer dans la liste-résultat en la maintenant triée.*

```
tri_insert : 'a list -> 'a list
```

▶ **MP & MP* – Option Informatique**
Premier TP Caml
Mardi 11 et 18 septembre 2007

# Mise en jambe CamlLight

## Un corrigé

### ▶ Question 1

```
let f = fun x -> x * x;;
let f = function x -> x * x;;
let f x = x * x;;

let rec g = fun 0 -> 1
    | x -> x * g (x-1);;
    (* autre possibilité *)
let rec g = fun x ->
  match x with
    | 0 -> 1
    | x -> x * g (x-1);;
let rec g = function 0 -> 1
    | x -> x * g (x-1);;
let rec g x =
   match x with
    0 -> 1
    | x -> x * g (x-1);;

let h = fun x y -> x + y;;
let h = function x -> function y -> x + y;;
let h x y = x + y;;
```

### ▶ Question 2

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

### ▶ Question 3

```
let compose = fun 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 = fun x -> 2 * x;;

(next o mul) 3;;
(mul o next) 3;;
```

### ▶ Question 4

```
let rec exp = fun x n -> match n with
    | 0 -> 1
    | _ -> x * (exp x (n-1))
;;
```

### ▶ Question 5

```
let rec exp = fun x n ->
    match n with
    | 0 -> 1
    | 1 -> x
    | _ when n mod 2 = 0 ->
        let b = exp x (n/2) in b * b
    | _ -> let b = exp x ((n-1)/2) in x * b * b
;;
```

### ▶ Question 6

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

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

### ▶ Question 7

```
let print_int_list = fun v -> iter print_int v;;
```

### ▶ Question 8

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

### ▶ Question 9

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

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

### ▶ Question 10

```
let rec half = fun 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 11

```
let rec fusion = fun 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 12

```
let rec tri_fusion = fun 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 13

```
let rec insert = fun l e ->
    match l with
    | [] -> e::[]
    | t::q when e < t -> e::l
    | t::q -> t::(insert q e)
;;
insert [2; 3; 7; 8] 9;;
```

## ▶ Question 14

```
let rec tri_insert = fun l ->
    match l with
    | [] -> []
    | t::q -> insert (tri_insert q) t
;;

tri_insert [5; 4; 3; 2; 8; 9; 0; 1];;
```