▶ **MPSI – Option Informatique**
Année 2002, Premier TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Suites récurrentes

Dans ce premier TP d'introduction, nous commençons par entrer quelques expressions Caml simples pour nous familiariser avec le système. Dans la deuxième partie, nous nous intéressons au calcul des termes d'une suite récurrente simple, la suite de Fibonacci. La première méthode de calcul « naïve » ne permet pas d'obtenir les valeurs des termes de la suite, car le temps de calcul nécessaire est trop important. Nous implanterons ensuite deux méthodes plus efficaces. Enfin, dans la dernière partie, nous écrivons quelques fonctions pour calculer les termes de suites récurrentes de la forme $u_{n+1} = f(u_n)$.

N.B. : *Les questions marquées ⋆ sont plus difficiles. Vous pourrez dans un premier temps les sauter et y revenir en fin de séance.*

1 Premières expressions Caml

1.1 Expressions

Entrons une première expression : saisissez par exemple `2+3;;` et validez par entrée (les deux points-virgules marquent la fin de la "phrase"). *Caml* vous répond normalement − : `int = 5` ce qui signifie que vous avez entré une valeur, qui est de type entier et qui vaut 5.
On peut écrire des expressions plus compliquées, *Caml* respecte les priorités habituelles des opérateurs. Essayez par exemple `(11 + 4) * 3 + 2 + 6 * 2 / 3;;`.

1.2 Définitions

De même qu'en mathématiques on écrit « soit s la somme des nombres 1, 2 et 3 », on écrit en *Caml* `let s = 1 + 2 + 3;;`. *Caml* vous répond que vous venez de définir un *identificateur* (`s` :) qui est un entier (`int`) et qui vaut 6. Vous pouvez maintenant utiliser s dans toutes vos expressions. Essayez par exemple `(2 + s) * s + 3;;` et `let p = 2 * s + s * s;;`.
Les définitions de noms que nous venons de voir sont permanentes : elles restent valides tant que vous n'abandonnez pas le système *Caml*. On dit qu'elles sont *globales*. On peut également faire des définitions locales à une phrase : `let y = 12 * 3 in 2 * y + y / 2;;`. Ici, y n'est défini que dans ce qui est entre le `in` et les `;;`. Si vous tapez maintenant `y;;`, *Caml* vous répond que cet identificateur n'est pas lié.

1.3 Fonctions

La syntaxe de la définition des fonctions en *Caml* est proche de la notation mathématique habituelle. Pour définir une fonction `carre` qui calcule le carré d'un réel x, on peut principalement utiliser les deux écritures suivantes qui sont équivalentes :

```
let carre x = x * x;;
let carre = function x -> x * x;;
```

Après avoir défini cette fonction, il vous suffit d'entrer `carre 11;;` pour calculer 11^2.
On définit très souvent en *Caml* des fonctions de manière récursive. Par exemple, on peut écrire une fonction qui, étant donnés deux entiers x et n, calcule x^n de manière récursive. Voici trois syntaxes possibles :

```
let rec pow x n =
  if n = 0 then 1
  else x * (pow x (n - 1))
;;

let rec pow x n =
  match n with
    0 -> 1
  | n -> x * (pow x (n - 1))
;;

let rec pow x = function
    0 -> 1
  | n -> x * (pow x (n - 1))
;;
```

© Vincent Simonet 2002
Ce document est distribué selon les termes de la GNU Free Documentation License.

Pour calculer 2^{10}, il suffit alors d'entrer pow 2 10.

▶ **Question 1** *Quel est le type de la fonction* pow *? Que représente l'expression* pow 2 *et quel est son type ?*

value pow : $int \to int \to int$

▶ **Question 2** *Combien de multiplications effectue-t-on pour calculer x^n en utilisant cette méthode ? Connaissez-vous une autre méthode de calcul plus efficace ?*

2 La suite de Fibonacci

La suite de Fibonacci est une suite d'entiers $(u_n)_{n\in\mathbb{N}}$ définie récursivement par les relations :

$$\begin{cases} u_0 = 0 \text{ et } u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

2.1 Calcul en temps exponentiel

On souhaite pouvoir calculer à l'aide de *Caml* n'importe quel terme de cette suite. *Caml* permettant de définir simplement des fonctions de manière récursive, une idée naturelle est de s'inspirer directement de la définition de la suite.

▶ **Question 3** *En utilisant la définition récursive de la suite $(u_n)_{n\in\mathbb{N}}$ donnée ci-dessus, programmez une fonction* fib *telle que* fib n *calcule u_n.*

value fib : $int \to int$

▶ **Question 4** *Calculez les premiers entiers de Fibonacci, puis calculez u_{40}. Comment expliquez-vous que le temps de calcul soit si long ?*

2.2 Calcul en temps linéaire

On peut effectuer un calcul plus efficace des termes de la suite en écrivant une fonction fib2 qui retourne une paire de deux termes consécutifs de la suite, c'est-à-dire telle que fib2 n donne la paire (u_n, u_{n+1}) : chaque appel de la fonction fib2 ne nécessitera alors plus qu'un seul appel récursif.

▶ **Question 5** *Programmez une telle fonction* fib2*. Combien d'additions effectue-t-on pour calculer u_n avec cette méthode ?*

value fib2 : $int \to int \times int$

2.3 Calcul en temps logarithmique

On peut vérifier que pour tous $n \geq 0$ et $p \geq 1$ on a

$$u_{p+q} = u_{p+1}u_q + u_p u_{q-1}$$

▶ **Question 6 (⋆)** *Exprimez u_{2m}, u_{2m+1} et u_{2m+2} en fonction de u_m et u_{m+1}.*

▶ **Question 7 (⋆)** *Déduisez-en une procédure* fibolog *telle que* fibolog n *calcule le couple (u_n, u_{n+1}). Vous pourrez distinguer deux cas suivant la parité de n.*

value fibolog : $int \to int \times int$

▶ **Question 8 (⋆)** *Évaluez le nombre d'appels récursifs effectué par cette nouvelle fonction. Comparez sa vitesse d'exécution avec celle de la précédente pour de très grandes valeurs de n.*

3 Suites « $u_{n+1} = f(u_n)$ »

Soit f une fonction d'un ensemble A dans lui-même et a un élément de A. On définit alors une suite $(u_n)_{n\in\mathbb{N}}$ à valeurs dans A par :

$$\begin{cases} u_0 = a \\ u_{n+1} = f(u_n) \end{cases}$$

Remarquons qu'une telle suite est définie par la donnée de la paire (f, a).

▶ **Question 9** *Écrivez une fonction* term *qui prend pour arguments la paire (f, a) et l'entier n et calcule u_n. Que représente en Caml l'expression* term (f , a) *?*

value term : $(\alpha \to \alpha) \times \alpha \to int \to \alpha$

On souhaite définir une fonction qui calcule la liste $[u_m; \ldots; u_0]$ des termes successifs de la suite. On pourrait pour cela écrire un algorithme qui appelle successivement la fonction term pour des valeurs de n variant de 0 à m. Cependant, cette méthode n'est pas efficace : pour chaque élément de la liste résultat, on reprend les calculs de zéro alors qu'il est facile de calculer un élément de la liste à partir de l'élément précédent.

▶ **Question 10** *En tenant compte de cette remarque, écrivez une fonction* list *telle que* list (f , a) m *retourne la liste $[u_m; \ldots; u_0]$.*

value list : $(\alpha \to \alpha) \times \alpha \to int \to \alpha\ list$

On veut maintenant obtenir les éléments dans l'ordre inverse. On propose pour cela deux méthodes différentes.

▶ **Question 11** *Écrivez une fonction* map *telle que* map f [x0 ; ...; xm] *retourne* [f x0 ; ...; f xm]*. Déduisez-en une fonction* list_rev *telle que* list_rev (f , a) n *calcule la liste $[u_0; \ldots; u_m]$*

value map : $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$
value list_rev : $(\alpha \to \alpha) \times \alpha \to int \to \alpha\ list$

▶ **Question 12** *Écrivez une fonction* rev *calculant le miroir d'une liste (i.e. telle que* rev [x1 ; ...; xn] *retourne* [xn ; ...; x1]*). Déduisez-en une autre implémentation de la fonction fonction* list_rev*. Quelle version vous semble la plus efficace ?*

value map : $\alpha\ list \to \alpha\ list$
value list_rev : $(\alpha \to \alpha) \times \alpha \to int \to \alpha\ list$

▶ **MPSI – Option Informatique**
Année 2002, Premier TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Suites récurrentes

Un corrigé

▶ **Question 1** La fonction pow prend deux entiers pour arguments et calcule un entier. Elle est de type $int \to int \to int$. L'expression pow 2 est une application partielle de la fonction pow. Elle représente la fonction $n \mapsto 2^n$. Elle est de type $int \to int$.

▶ **Question 2** On vérifie par récurrence que pour calculer x^n ces fonctions effectuent exactement n multiplications. Il est possible de calculer x^n en effectuant un nombre de multiplications dominé par le logarithme de n (algorithme d'exponentiation rapide).

▶ **Question 3** Voici trois possibilités équivalentes (reprenant des syntaxes analogues à celles utilisées pour la fonction pow).

```
let rec fib n =
  if n <= 1 then n
  else fib (n - 1) + fib (n - 2)
;;

let rec fib n =
  match n with
    0 -> 0
  | 1 -> 1
  | n -> fib (n - 1) + fib (n - 2)
;;

let rec fib = function
    0 -> 0
  | 1 -> 1
  | n -> fib (n - 1) + fib (n - 2)
;;
```

▶ **Question 4** Pour calculer u_{40} notre fonction calcule u_{39} puis u_{38}, bien que l'on ait déjà calculé cet entier lors du calcul de u_{39}. Les mêmes calculs sont donc répétés plusieurs fois de manière inutile.
Notons $c(n)$ le nombre de sommes d'entiers effectué par la fonction fib pour calculer u_n. On a $c(0) = c(1) = 0$ et $c(n) = c(n-1)+c(n-2)+1$. Cette relation de récurrence admet pour solution

$$c(n) = -1 + \frac{1}{\sqrt{5}} \left(\left(\frac{2}{\sqrt{5}-1} \right)^{n+1} - \left(-\frac{2}{\sqrt{5}+1} \right)^{n+1} \right)$$

On constate que le coût du calcul croît exponentiellement avec n, ce qui rend la fonction rapidement inutilisable.

▶ **Question 5**

```
let rec fib2 = function
    0 -> (0, 1)
  | n ->
    let (x, y) = fib2 (n - 1) in
    (y, x + y)
;;
```

Avec cette méthode, on peut calculer u_n (pour $n \geq 1$) en effectuant $n - 1$ additions.

▶ **Question 6** On obtient facilement les expressions voulues en utilisant l'égalité donnée par l'énoncé ainsi que la relation de récurrence définissant la suite :

$$\begin{aligned}
u_{2m} &= u_{m+m} \\
&= u_{m+1}u_m + u_m u_{m-1} \\
&= u_{m+1}u_m + u_m(u_{m+1} - u_m) \\
&= 2u_{m+1}u_m - u_m^2 \\
u_{2m+1} &= u_{m+(m+1)} \\
&= u_{m+1}u_{m+1} + u_m u_m \\
&= u_{m+1}^2 + u_m^2 \\
u_{2m+2} &= u_{(m+1)+(m+1)} \\
&= u_{m+2}u_{m+1} + u_{m+1}u_m \\
&= (u_{m+1} + u_m)u_{m+1} + u_{m+1}u_m \\
&= u_{m+1}^2 + 2u_{m+1}u_m
\end{aligned}$$

▶ **Question 7** Notre fonction distingue trois cas : le cas $n = 0$, le cas n pair (pour tester si n est pair, il suffit de calculer le reste de la division euclidienne de n par 2, $n \mod 2$, et de le comparer à 0) et le cas où n est impair (il est inutile de mettre un garde **when** (n mod 2) = 1 pour ce dernier cas car *Caml* choisi toujours le premier cas qui convient dans un filtrage).

```
let rec fibolog = function
  0 -> (0, 1)
| n when (n mod 2) = 0 ->
  let (a, b) = fibolog (n / 2) in
  (2 * b * a - a * a, b * b + a * a)
| n ->
  let (a, b) = fibolog ((n - 1) / 2) in
  (b * b + a * a, b * b + 2 * b * a)
;;
```

► **Question 8** Notons $c(n)$ le nombre d'appels récursifs effectués lors de l'appel de `fibolog n`. Nous allons calculer $c(n)$ dans le cas où n est une puissance de 2 (i.e. $n = 2^m$).
On a $c(2^0) = 1$. Le calcul de u_{2^m} fait un appel récursif au rang 2^{m-1}. On a donc $c(2^m) = c(2^{m-1})+1$. On en déduit que $c(2^m) = m + 1$ ce qui nous donne $c(n) = \log_2 n + 1$ (dans le cas où n est une puissance de 2).
On peut vérifier que cet « ordre de grandeur » est respecté même dans le cas où n n'est pas une puissance de 2. On dit que notre algorithme a une complexité logarithmique en n.

► **Question 9**

```
let rec term (f, a) n =
  match n with
    0 -> a
  | n -> f (term (f, a) (n - 1))
;;
```

Cette première version n'est pas *récursive terminale* : dans le cas $n \neq 0$, l'appel récursif n'est pas la dernière opération effectuée : il faut ensuite appliquer f au résultat donné par l'appel récursif. On peut donc préférer la version suivante qui est récursive terminale :

```
let rec term (f, a) n =
  match n with
    0 -> a
  | n -> term (f, (f a)) (n - 1)
;;
```

Dans ce cas, l'appel récursif est la dernière opération effectuée. De manière générale, les fonctions récursives terminales sont plus efficaces car le compilateur peut « oublier » l'endroit où l'appel récursif est effectué puisqu'il suffit de retourner directement le résultat de cet appel.

L'expression `term (f, a)` représente exactement la « fonction » $n \mapsto u_n$.

► **Question 10**

```
let rec list (f, a) = function
    0 -> [a]
  | m ->
    let q = list (f, a) (m - 1) in
    (f (hd q)) :: q
;;
```

On utilise dans cette la fonction `hd` de la bibliothèque standard qui retourne la tête d'une liste (i.e. son premier élément).

► **Question 11**

```
let rec map f = function
    [] -> []
  | t :: q -> (f t) :: (map f q)
;;

let rec list_rev (f, a) = function
    0 -> [a]
  | m -> a :: (map f (list_rev (f, a) (m - 1)))
;;
```

Chaque appel à la fonction `map` effectue un nombre d'opérations proportionnel à la longueur de son argument. De plus, le calcul de `list_rev (f, a) m` appelle la fonction `map` sur des listes de longueurs successives $1, \ldots, m-1$. On en déduit que le temps de calcul est quadratique en la valeur de m.

► **Question 12**

```
let rec rev = function
    [] -> []
  | t :: q -> (rev q) @ [t]
;;

let rec list_rev (f, a) m =
  rev (list (f, a) m)
let rec list_rev (f, a) = function
    0 -> [a]
  | m -> a :: (map (list_rev (f, a) (m - 1)))
;;
```

Notre implémentation « naïve » de la fonction `rev` a une complexité quadratique en la taille de la liste passée en argument. Il est cependant possible d'écrire une version plus efficace de cette fonction :

```
let rec rev_append list1 list2 =
  match list1 with
    [] -> list2
  | t :: q -> rev_append q (t :: list2)
;;

let rev list = rev_append list [];;
```

La fonction intermédiaire `rev_append` effectue la concaténation du miroir d'une liste avec une seconde liste. Elle a un temps d'exécution linéaire en la taille de la première liste.

Memento

Vincent.Simonet@inria.fr
http://cristal.inria.fr/~simonet/teaching/

La page du langage Caml : http://caml.inria.fr/