▶ **MPSI – Option Informatique**
Année 2001, 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 deux fonctions génériques permettant de calculer les termes de suites récurrentes linéaires ou de la forme* $u_{n+1} = f(u_n)$.

## 1 Premières expressions Caml

### 1.1 Premières 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 (`int`) et qui vaut 5 (`= 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 Les 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 qui disparaissent à la fin de l'évaluation de la phrase dans laquelle elles se trouvent : `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 Premières 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 puiss x n =
  if n = 0 then 1
  else x * (puiss x (n - 1))
;;

let rec puiss x n =
```

```
  match n with
    0 -> 1
  | n -> x * (puiss x (n - 1))
;;

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

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

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

▶ **Question 2** *Combien de multiplications effectue-t-on pour calculer $x^n$ en utilisant cette méthode ? En supposant que la multiplication est définie de la même manière à partir de l'addition, combien de sommes cela fait-il ? 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 récursif des termes de la suite

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* fibrec *telle que* fibrec n *calcule $u_n$ pour tout entier $n$. Vous écrirez trois versions analogues à celles données pour la fonction* puiss*.*

▶ **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 itératif des termes de la suite

On veut maintenant calculer tous les termes de la suite de Fibonnaci de rangs compris entre 0 et un entier $n$ donné.

▶ **Question 5** *Programmez une fonction* fibit *telle que* fibit n *retourne un tableau de $n+1$ entiers dont la case d'index $i$ contienne le terme de rang $i$ de la suite.*

▶ **Question 6** *Combien d'additions effectue-t-on pour calculer avec cette méthode $u_n$ ? Quel espace mémoire utilise-t-on ? Pourriez-vous faire mieux ?*

## 2.3 Une dernière méthode plus rapide

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 7** *Exprimez $u_{2m}$, $u_{2m+1}$ et $u_{2m+2}$ en fonction de $u_m$ et $u_{m+1}$.*

▶ **Question 8** *Déduisez-en une procédure* fibolog *telle que* fibolog n *calcule le couple $(u_n, u_{n+1})$. Vous pourrez distinguer deux cas dans votre fonction suivant la parité de $n$.*

▶ **Question 9** *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 $n$.*
*Pour certaines valeurs de $n$, Caml donne un résultat négatif, pourquoi ?*

# 3 D'autres suites récurrentes

On s'intéresse dans cette question au calcul de suites définies par des relations de récurrence de deux formes particulières usuelles.

## 3.1 Suites $u_{n+1} = f(u_n)$

Soit $f$ une fonction de $\mathbb{Z}$ dans lui-même et un entier $a$. On définit alors une suite $(u_n)_{n\in\mathbb{N}}$ par :

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

▶ **Question 10** *Écrivez une fonction $S$ qui prend pour arguments $f$, $a$ et $n$ et calcule $u_n$.*
*Écrivez ensuite une fonction $T$ telle que* T f a *retourne la fonction $u$.*

## 3.2 Suites récurrentes linéaires

Étant donnés deux vecteurs d'entiers $a = [|a_0, \ldots, a_{k-1}|]$ et $b = [|b_0, \ldots, b_{k-1}|]$, on peut définir une suite $(u_n)_{n\in\mathbb{N}}$ par :

$$\begin{cases} u_0 = a_0, \ldots, u_{k-1} = a_{k-1} \\ u_n = b_0 u_{n-k} + b_1 u_{n-k+1} + \ldots + b_{k-1} u_{n-1} \end{cases}$$

▶ **Question 11** *Écrivez une fonction qui prend pour argument $b$ et le vecteur $[|u_{n-k}, ..., u_{n-1}|]$ et retourne l'entier $u_n$.*
*Déduisez-en une fonction qui prend pour argument $b$ et $[|u_{n-k}, ..., u_{n-1}|]$ et retourne le vecteur suivant, $[|u_{n-k+1}, ..., u_n|]$.*

▶ **Question 12** *Programmez enfin une fonction* reclin *telle que* reclin a b n *calcule $u_n$. Évaluez, en fonction de $n$ et $k$, le coût du calcul de $u_n$ par cette méthode.*

▶ **MPSI – Option Informatique**
Année 2001, Premier TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

# Suites récurrentes

## Un corrigé

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

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

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

let rec fibrec = function
    0 -> 0
  | 1 -> 1
  | n -> fibrec (n - 1) + fibrec (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 fibrec 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** Notre fonction comporte quatre étapes successives :

1. Création d'un tableau $t$ de $n+1$ cases à l'aide de la fonction make_vect et d'une définition locale **let** ... **in** ...,
2. Initialisation des cases du tableau d'index 0 et 1,
3. Calcul à l'aide d'une boucle **for** des éléments du tableau de proche en proche,
4. Renvoi du tableau $t$.

```
let fibit n =
  let t = make_vect (n + 1) 0 in
  t.(0) <- 0;
  t.(1) <- 1;
  for i = 2 to n do
    t.(i) <- t.(i - 1) + t.(i - 2)
  done;
  t
;;
```

▶ **Question 6** Avec cette méthode, on calcule $u_n$ en effectuant $n-1$ additions. Cependant on crée un tableau de $n+1$ cases pour effectuer ce calcul. L'espace mémoire auxilliaire utilisé est donc linéaire en $n$. On peut améliorer ce point en se limitant à deux références auxilliaires :

```
let fibit' = function
  0 -> 0
| 1 -> 1
| n ->
  let a = ref 0
  and b = ref 1
  in
  for i = 2 to n do
  let c = !a + !b in
    a := !b;
    b := c
  done;
  !b
;;
```

▶ **Question 7** 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 8** 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 9** 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 10**

```
let rec S f a n =
  match n with
    0 -> a
  | n -> f (S 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 S f a n =
  match n with
    0 -> a
  | n -> S 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.

La fonction $T$ est en fait la fonction $S$. Cela est confirmé par leur type :

$$S : \underbrace{(int \to int)}_{f} \to \underbrace{int}_{a} \to \underbrace{int}_{n} \to \underbrace{int}_{u_n}$$

$$T : \underbrace{(int \to int)}_{f} \to \underbrace{int}_{a} \to \underbrace{(int \to int)}_{u}$$

▶ **Question 11**

```
let somme b t =
  let k = vect_length b in
  let s = ref 0 in
  for i = 0 to k - 1 do
    s := !s + b.(i) * t.(i)
  done;
  !s
;;

let next b t =
  let k = vect_length b in
  let t' = make_vect k 0 in
  for i = 0 to k - 2 do
    t'.(i) <- t.(i + 1)
  done;
  t'.(k - 1) <- somme b t;
  t'
;;
```

▶ **Question 12**

```
let reclin a b n =
  let rec aux a b = function
    0 -> a
  | n -> next b (aux a b (n - 1))
  in
  let k = vect_length a in
  if n < k then a.(n)
  else (aux a b (n - k + 1)).(k - 1)
;;
```

On vérifie facilement que le coût du calcul de $u_n$ est un $O(nk)$.