CAML LIGHT 2 - RÉCURSIVITÉ

1. Quatre exemples [1]

1.1. **Factorielle.** Rappelons que la factorielle de l'entier naturel n est, par définition, l'entier :

$$n! = \prod_{k=1}^{n} k$$

avec la convention $0! = 1$.

La factorielle d'un entier naturel non nul s'exprime en fonction de celle de l'entier précédent :

$$\forall n \in \mathbb{N}^\star,\ n! = n\ (n-1)!$$

Ceci suggère de définir la factorielle "en fonction d'elle-même", c'est-à-dire **récursivement** :

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
;;
fact : int -> int = <fun>
```

On obtient par exemple :

```
fact 5;;
- : int = 120
```

Remarquer la présence du mot-clef `rec`, indiquant que la fonction `fact` est définie récursivement. Sans lui, CAML renvoie un message d'erreur :

```
let fact n =
  if n = 0 then 1 else n * fact (n-1)
;;
Toplevel input:
>  if n = 0 then 1 else n * fact (n-1)
>                          ^^^^
The value identifier fact is unbound.
```

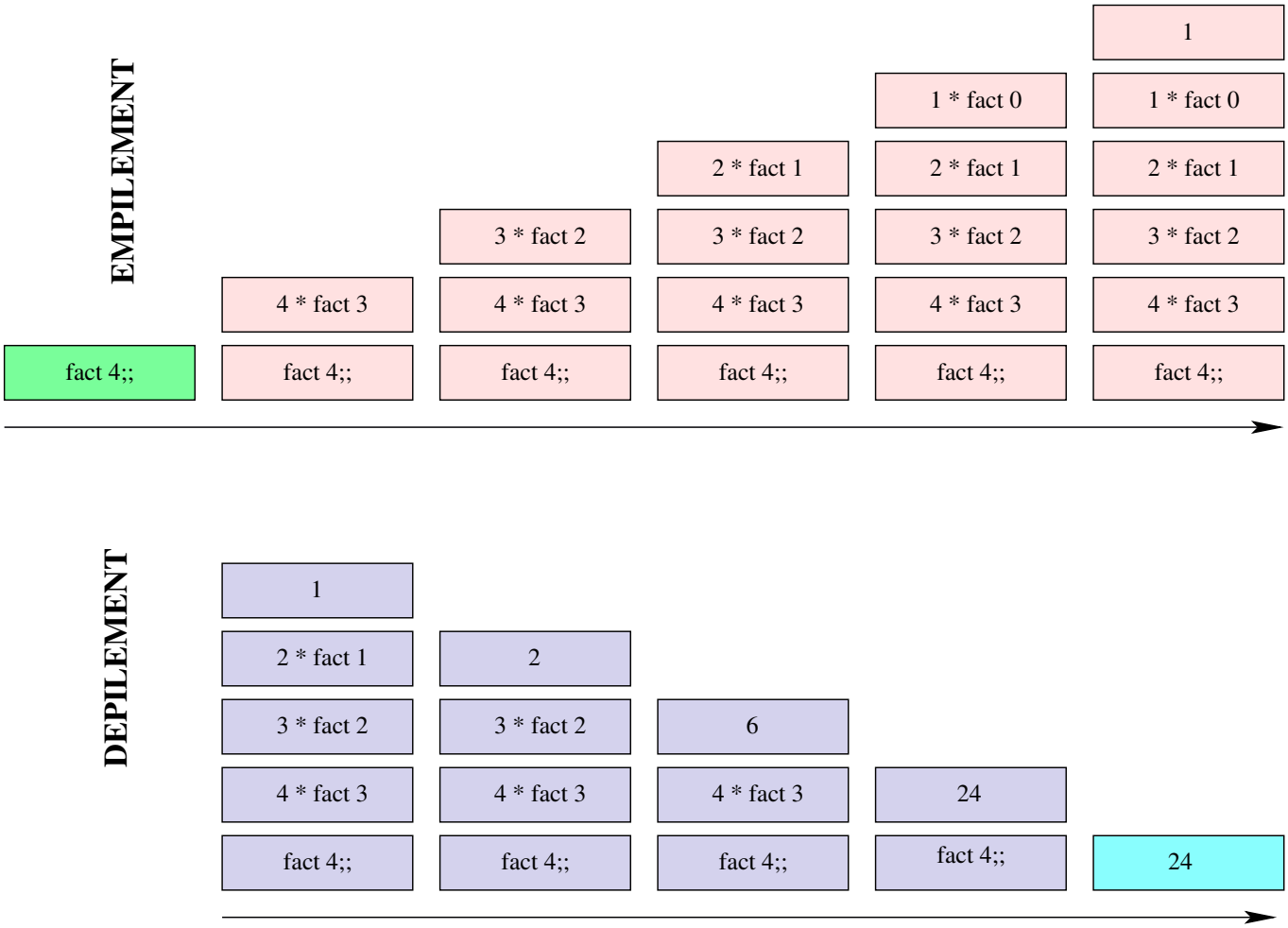
Tout ceci peut paraître un peu "magique", mais le concept de récursivité est fondamental et doit devenir – à l'usage – familier et naturel.

Comment CAML évalue-t-il l'expression `fact 4` ? Cet appel renvoie dans un premier temps `4 * fact 3`.

CAML mémorise qu'il devra multiplier 4 par un résultat non encore disponible : on dit qu'il y a **empilement du contexte**. A son tour, l'expression `fact 3` est évaluée en `3 * fact 2` et le contexte est empilé ; puis `fact 2` est évaluée en `2 * fact 1` et le contexte est empilé, puis `fact 1` est évaluée en `1 * fact 0` et le contexte est empilé. Enfin, `fact 0` est directement évaluée : on dit qu'il s'agit d'un **cas de base** (c'est-à-dire : une valeur pour laquelle la fonction est évaluée directement, sans faire appel à elle-même). Mais ce n'est pas encore fini : il faut à présent dépiler !

Le diagramme suivant schématise l'ensemble du mécanisme :

1. Un exemple graphique (courbe de Von Koch) sera présenté dans le document Caml Light 3.



Nous verrons plus loin comment la fonction `fact` ci-dessus peut être améliorée (voir section 2 : récursivité terminale).

1.2. **Suite de Fibonacci.** Il s'agit de la suite définie par la donnée de ses deux premiers termes $F_0 = 0$, $F_1 = 1$ et la relation de récurrence (linéaire du second ordre, à coefficients constants) :

$$\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$$

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib(n-1) + fib(n-2)
;;
fib : int -> int = <fun>
```

Sur cet exemple, on voit encore à quel point la syntaxe de CAML suit de près la notation mathématique. Cependant, ce code comporte un inconvénient majeur ...

En effet, si l'on note q_n le nombre d'appels récursifs déclenchés par l'évaluation de `fib n`, alors $q_0 = q_1 = 0$ et pour tout $n \geqslant 2$: $q_n = 2 + q_{n-1} + q_{n-2}$. On montre aisément par récurrence ($\rightarrow$ le faire !) que : $\forall n \in \mathbb{N},\ q_n = 2\left(F_{n+1} - 1\right)$. Pour évaluer l'expression `fib n`, la fonction `fib` est donc appelée plus de fois que ce qu'elle calcule ! Pour être précis, si l'on note g le nombre d'or $\left(g = \dfrac{1+\sqrt{5}}{2} \simeq 1,618\right)$, alors $q_n \sim \dfrac{2}{\sqrt{5}} g^{n+1}$ lorsque $n \to \infty$.

La complexité (évaluée ici en nombre d'appels récursifs) est donc exponentielle ...

On pourrait aussi mesurer la complexité en s'intéressant au nombre s_n d'additions engendrées par l'évaluation de `fib n`. Alors $s_0 = s_1 = 0$ et $s_n = 1 + s_{n-1} + s_{n-2}$. Donc : $\forall n \in \mathbb{N},\ s_n = F_{n+1} - 1 \sim \dfrac{1}{\sqrt{5}} g^{n+1}$.

On peut remédier à cela, et de plusieurs façons (voir TD et plus bas).

1.3. **Puissances d'un entier.** Le calcul récursif des puissances d'un entier peut se faire comme suit :

```
let rec puissance x n =
  if n = 0 then 1 else x * (puissance x (n-1))
;;
puissance : int -> int -> int = <fun>
```

Cet algorithme naïf n'est toutefois pas très performant. La remarque qui suit donne l'idée d'une meilleure méthode.

Supposons par exemple que l'on veuille calculer 2^{20} :

$$2^{20} = \left(2^{10}\right)^2 = \left(\left(2^5\right)^2\right)^2 = \left(\left(2\times 2^4\right)^2\right)^2 = \left(\left(2\times\left(2^2\right)^2\right)^2\right)^2$$

L'évaluation de la dernière expression écrite requiert 5 multiplications (au lieu de 19 pour la méthode naïve).

Plus généralement, on s'appuie sur le fait que :

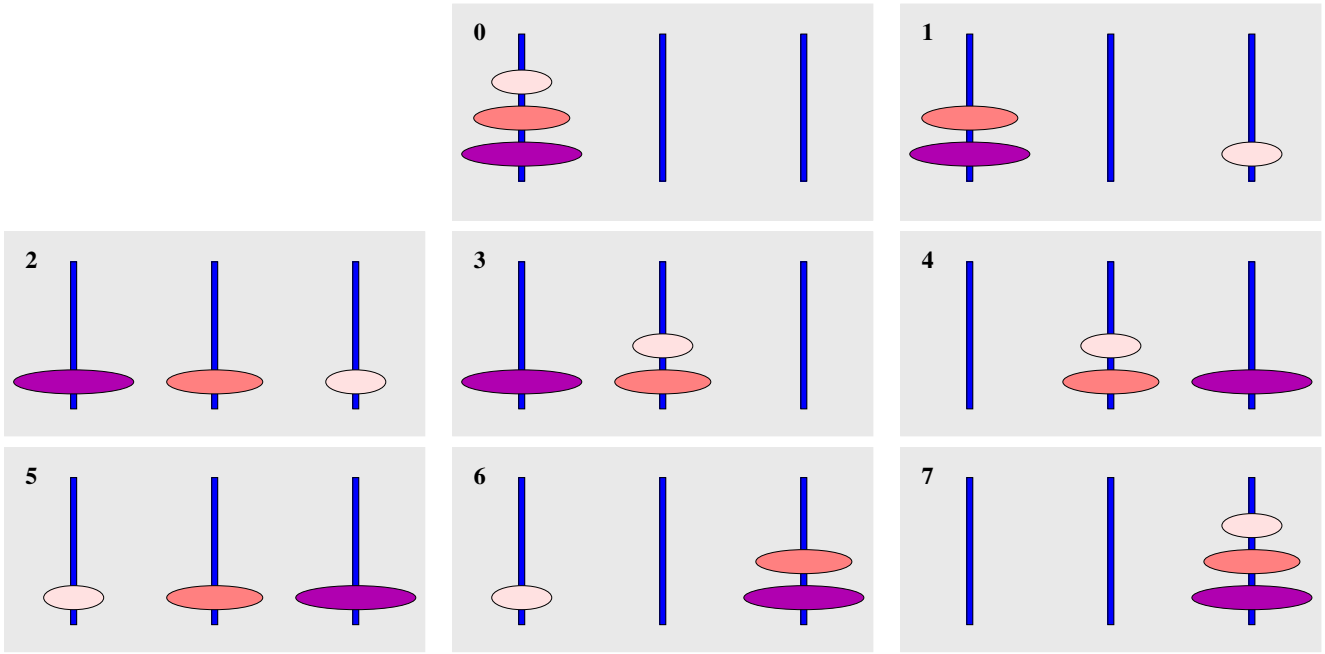
$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ x\left(x^{\frac{n-1}{2}}\right)^2 & \text{sinon} \end{cases}$$

ce qui conduit à la méthode, dite d'EXPONENTIATION RAPIDE. Voici son implémentation en CAML :

```
let rec puiss_rapide x n =
  if n = 0 then 1 else
    let p = puiss_rapide x (n/2) in
      if n mod 2 = 0 then p*p else x*p*p
;;
puiss_rapide : int -> int -> int = <fun>
```

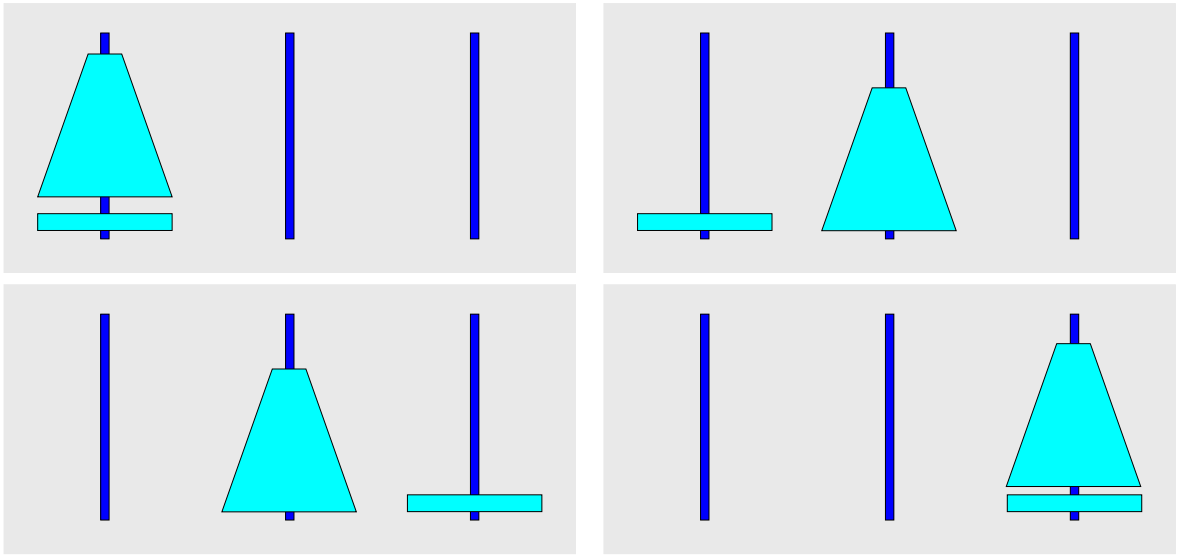
Nous nous intéresserons plus loin au calcul détaillé du nombre de multiplications requis pour calculer x^n par cette méthode.

1.4. **Tours de Hanoï.** On dispose de trois piquets. Sur le piquet de gauche, sont empilés n disques de diamètres tous distincts, par ordre décroissant de taille. On peut déplacer ces disques un à un, d'un piquet à un autre, en respectant une règle unique : on ne doit jamais placer un disque au-dessus d'un disque de diamètre inférieur. Le jeu consiste à déplacer tous les disques vers le piquet de droite.



Pour déplacer n disques du piquet a vers le piquet b en utilisant le piquet i comme intermédiaire, on va :

(1) Déplacer $n-1$ disques de a vers i en utilisant b comme intermédiaire
(2) Déplacer 1 disque de a vers b
(3) Déplacer $n-1$ disques de i vers b en utilisant a comme intermédiaire



La "traduction" en CAML est alors très naturelle (et récursive !) :

```
let rec hanoi n a b i =
  if n = 1 then (
    print_int a;
    print_string " -> ";
    print_int b;
    print_newline ();
  )
  else (
    hanoi (n-1) a i b;
    hanoi 1 a b i;
    hanoi (n-1) i b a
  )
;;
hanoi : int -> int -> int -> int -> unit = <fun>
```

A titre d'exemple :

```
hanoi 4 1 3 2;;
1 -> 2
1 -> 3
2 -> 3
1 -> 2
3 -> 1
3 -> 2
1 -> 2
1 -> 3
2 -> 3
2 -> 1
3 -> 1
2 -> 3
1 -> 2
1 -> 3
2 -> 3
- : unit = ()
```

Il peut être amusant de modifier la fonction `hanoi` pour que l'affichage de la solution soit graphique ... et animé ! On trouve sur le net de telles animations, par exemple :

`http://thinks.com/java/hanoi/hanoi.htm`

On peut montrer que le nombre minimal de mouvements nécessaires pour résoudre ce problème est $2^n - 1$.

Le jeu des tours de Hanoï trouve son origine dans un ouvrage écrit par le mathématicien français Edouard LUCAS (1842-1891), intitulé *Récréations Mathématiques* et publié vers la fin du XIX$^{\text{ème}}$ siècle. L'auteur y invente une légende selon laquelle des bonzes passent leur temps à résoudre ce problème pour $n = 64 \cdots$ et la fin du monde surviendra ensuite. En admettant que le déplacement de chaque disque prenne une seconde, ça nous laisse encore un peu de temps ... car :

$$2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$$

ce qui représente tout de même un peu plus de $5,8 \times 10^{11}$ années.

2. Récursivité terminale

• Revenons au calcul de la factorielle d'un entier. Si l'on reprend (cf. section 1.1), avec filtrage de motif[2] pour changer :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1)
;;
fact : int -> int = <fun>
```

alors, l'évaluation de `fact n` conduit, par empilement du contexte, à une occupation mémoire qui est en gros proportionnelle à `n`.

En revanche, si l'on adopte :

```
let fact n = f 1 n where
  rec f a = fun
    | 0 -> a
    | n -> f (n*a) (n-1)
;;
fact : int -> int = <fun>
```

alors l'occupation mémoire est constante, comme pour un programme itératif. Cette fonction est dite **récursive terminale** ("tail recursive" en anglais).

Schématiquement :

`fact 5` $\rightarrow$ `f 1 5` $\rightarrow$ `f 5 4` $\rightarrow$ `f 20 3` $\rightarrow$ `f 60 2` $\rightarrow$ `f 120 1` $\rightarrow$ `f 120 0` $\rightarrow$ `120`

Conceptuellement :

$$f(x) = si\ R(x)\ alors\ g(x)\ sinon\ h(x, f(k(x)))$$

est la définition d'une fonction récursive non terminale : si la condition $R(x)$ est remplie, alors le calcul se termine avec l'évaluation de la fonction g; dans le cas contraire, on effectue un appel récursif à f qui, lorsqu'il sera terminé, n'achèvera pas le calcul car il faudra encore évaluer $h(x, \cdots)$. Chaque appel à f est associé à une zone en mémoire où l'ensemble du contexte (à savoir les paramètres qui ont été passés à f ainsi que l'endroit où l'appel s'est interrompu) est conservé : chacun des appels récursifs peut ainsi se terminer.

En revanche :

$$f(x) = si\ R(x)\ alors\ g(x)\ sinon\ f(k(x))$$

est la définition d'une fonction récursive terminale. Lorsque l'appel $f(k(x))$ se termine, alors l'appel précédent se termine aussi.

---

2. Le filtrage de motif est présenté dans un document ultérieur : prière de s'y reporter ...

Il faut savoir que CAML "reconnaît" qu'une fonction est récursive terminale ; l'exécution du code compilé est (presque) aussi efficace que pour une boucle.

• Autre exemple : reprenons le calcul des puissances, vu plus haut. La procédure `puissance` était récursive non terminale ; en voici une version récursive terminale :

```
let puissance x n = puiss 1 n where
  rec puiss p n =
  if n = 0 then p else puiss (p*x) (n-1)
;;
puissance : int -> int -> int = <fun>
```

ou bien avec filtrage de motif :

```
let puissance x n = puiss 1 n where
  rec puiss p = function
    | 0 -> p
    | n -> puiss (p*x) (n-1)
;;
puissance : int -> int -> int = <fun>
```

Schématiquement :

`puissance 3 4` → `puiss 1 4` → `puiss 3 3` → `puiss 9 2` → `puiss 27 1` → `puiss 81 0` → `81`

• Dernier exemple, le calcul des termes de la suite de Fibonacci, version récursive terminale :

```
let fib n = f 0 1 n where
  rec f a b = function
    | 0 -> a
    | 1 -> b
    | n -> f b (a+b) (n-1)
;;
fib : int -> int = <fun>
```

Schématiquement :

`fib 8` → `f 0 1 8` → `f 1 1 7` → `f 1 2 6` → `f 2 3 5`
→ `f 3 5 4` → `f 5 8 3` → `f 8 13 2` → `f 13 21 1` → `21`

On a décrit dans ce document quelques exemples simples de fonctions récursives en CAML. De nombreux autres exemples seront examinés par la suite, notamment lors de l'étude des listes.

3. Dérécursification

Disons simplement qu'à tout algorithme récursif, il est possible d'associer un algorithme itératif équivalent. Pour cela, ce dernier doit se doter d'une (ou de plusieurs) pile(s) afin de simuler le stockage des arguments et des valeurs de retour. Ce processus est appelé "dérécursification". Dans le cas d'une récursion terminale, la dérécursification ne nécessite l'emploi d'aucune pile.