

module 4 :

Itération

Durant ce module, nous allons confronter deux notions : la notion mathématique d'*itération* et la notion informatique de *boucle*.

J'appelle *itération* le calcul de l'élément d'indice k d'une suite $(u_n)_{n \in \mathbb{N}}$ définie par la donnée de u_0 et d'une relation de récurrence : $u_{n+1} = f(u_n)$.

J'appelle *boucle* toute description algorithmique d'une instruction qui possède : une entrée, un corps qui accomplit une action, un circuit de répétition qui répète l'exécution du corps, et une sortie.

Une boucle est dite *indexée* lorsque le nombre de fois que le corps de la boucle doit être répété est connu à l'entrée de la boucle.

Une boucle est dite *conditionnelle* lorsque le corps de la boucle est exécutée tant qu'une condition est vérifiée.

Le théorème sur lequel nous allons nous appuyer est le suivant :

Une boucle peut toujours être décrite par une itération.

En effet, si d est l'ensemble des données manipulées dans le corps de la boucle, d_0 l'état initial de ces données et f la transformation qui affecte ces données dans le corps de la boucle, effectuer cette boucle revient à itérer la suite définie par d_0 et la relation : $d_{n+1} = f(d_n)$.

Nous allons maintenant rentrer dans le détail des différentes boucles.

1. Boucles indexées

Elles se traduisent en Caml par la syntaxe : `for i = a to b do séquence d'instructions done.`

i est l'*indice* de la boucle. Il désigne une valeur entière qui varie entre les entiers a et b en augmentant de 1 à chaque itération.

Si $a \leq b$, le corps de la boucle est parcouru $b - a + 1$ fois ; si $a > b$, le corps de la boucle n'est jamais parcouru.

Exemple. Calcul de a^n lorsque a et n sont deux entiers :

```
let puissance a n =
  let x = ref 1 in
  for i = 1 to n do x := !x * a done ;
  !x ;;
```

L'itérateur correspondant est ici : $\forall n \in \mathbb{N}, u_n = a^n$, soit $u_0 = 1$ et $f : x \mapsto ax$.

Remarque. Cet algorithme nécessite n multiplications, soit un coût linéaire. On verra plus loin un algorithme ayant un coût logarithmique.

Exemple. Calcul de $n!$:

```
let fact n =
  let x = ref 1 in
  for i = 1 to n do x := !x * i done ;
  !x ;;
```

Les données manipulées sont i et $!x$. L'itérateur correspondant est donc ici : $\forall n \in \mathbb{N}, u_n = (n, n!)$, soit $u_0 = (0, 1)$ et $f : (x, y) \mapsto (x + 1, (x + 1)y)$.

• Un itérateur générique

Sachant qu'une boucle indexée se ramène à une itération, on peut commencer par écrire un itérateur générique :

```
let rec itère f u = fonction
  0 -> u
  | n -> f (itère f u (n-1)) ;;
```

ou

```
let rec itère f u = fonction
  0 -> u
  | n -> itère f (f u) (n-1) ;;
```

Son type est : $('a \rightarrow 'a) \rightarrow 'a \rightarrow \text{int} \rightarrow 'a$.

Les fonctions précédentes peuvent alors s'écrire :

```
let puissance a = itère (function x -> a*x) 1 ;;

let fact n =
  let (_,result) = itère (function (x,y) -> (x+1,(x+1)*y)) (0,1) n
  in result ;;
```

Notons que la fonction `itère` est définie par un algorithme récursif. Ceci permet, conjointement au théorème cité au début de ce texte, d'énoncer le résultat suivant :

Tout algorithme utilisant une boucle indexée possède une version récursive.

Pour obtenir la version récursive d'un algorithme utilisant une boucle indexée, il suffit donc de déterminer l'itérateur associé à cette boucle. Par exemple, on obtiendra pour les deux fonctions précédentes :

```
let rec puissance a = function
  0 -> 1
  | n -> a * puissance a (n-1) ;;

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

2. Boucles conditionnelles

On peut dégager deux types de boucles conditionnelles :

- tant que la condition est vérifiée, effectuer le corps de la boucle ;
- effectuer le corps de la boucle jusqu'à ce que la condition soit vérifiée.

Dans le premier cas, la condition est une condition d'entrée, dans le deuxième cas, une condition de sortie.

Une boucle du premier type peut ne pas exécuter le corps de la boucle (si la condition n'est pas vérifiée en entrée). Par contre, une boucle du deuxième type va effectuer au moins une fois le corps de la boucle avant de tester la condition.

Seul le premier type est implémenté par Caml suivant la syntaxe : `while condition do séquence d'instructions done`.

À l'instar des boucles indexées, une boucle conditionnelle se ramène à une itération. De ce fait, on peut écrire une fonction `tant_que` générique de la manière suivante :

```
let rec tant_que cond f u = match (cond u) with
  true -> tant_que cond f (f u)
  | false -> u ;;
```

Il s'agit d'une fonction de type $('a \rightarrow \text{bool}) \rightarrow ('a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$. Elle retourne le premier élément de la suite $(u_n)_{n \in \mathbb{N}}$ définie par $\begin{cases} u_0 = u \\ u_{n+1} = f(u_n) \end{cases}$ ne vérifiant pas la condition. On peut donc énoncer :

Tout algorithme utilisant une boucle conditionnelle possède une version récursive.

Remarque. On peut procéder de la même manière pour le deuxième type de boucle conditionnelle, à partir de l'itérateur générique suivant :

```
let répète cond f u = tant_que (function b -> not cond b ) f (f u) ;;
```

• Preuve de validité d'un algorithme

Démontrer la validité d'un algorithme utilisant une boucle indexée ou conditionnelle consiste à déterminer l'itérateur $(u_n)_{n \in \mathbb{N}}$ associé à cette boucle : on extrait de l'algorithme la valeur de u_0 ainsi que la relation de récurrence : $u_{k+1} = f(u_k)$, ce qui permet ensuite d'en déduire le résultat de l'algorithme, à savoir u_n (la détermination de la suite $(u_n)_{n \in \mathbb{N}}$ est ce qu'on appelle un *invariant de boucle*).

Exemple. Considérons l'algorithme suivant :

```
let exp a n =
  let x = ref 1 and y = ref a and z = ref n in
  while !z > 0 do
    if !z mod 2 = 1 then x := !x * !y ;
    z := !z / 2 ;
    y := !y * !y done ;
  !x ;;
```

Que calcule-t-il? Pour répondre à cette question, nous allons déterminer l'invariant de boucle, qui ici est un triplet (x_k, y_k, z_k) défini par les valeurs initiales : $(x_0 = 1, y_0 = a, z_0 = n)$ et les relations :

$$x_{k+1} = \begin{cases} x_k y_k & \text{si } z_k \text{ est impair} \\ x_k & \text{sinon} \end{cases}, \quad y_{k+1} = y_k^2, \quad z_{k+1} = \lfloor \frac{z_k}{2} \rfloor.$$

On obtient facilement : $\forall k \in \mathbb{N}, y_k = a^{(2^k)}$.

Pour déterminer z_k , nous allons introduire la décomposition en base 2 de l'entier n :

$$n = [b_p, b_{p-1}, \dots, b_1, b_0]_2 \stackrel{\text{def}}{=} b_p 2^p + b_{p-1} 2^{p-1} + \dots + b_0.$$

Il apparaît alors que pour tout $k \in \llbracket 0, p \rrbracket$, $z_k = [b_p, b_{p-1}, \dots, b_k]_2$.

À cette étape de la démonstration, on peut affirmer que $z_p = b_p \neq 0$ et $z_{p+1} = 0$, ce qui prouve déjà que cet algorithme se termine et retourne la valeur de x_{p+1} .

Remarquons enfin que la formule : $x_{k+1} = \begin{cases} x_k y_k & \text{si } b_k = 1 \\ x_k & \text{si } b_k = 0 \end{cases}$ se réduit à : $x_{k+1} = x_k y_k^{b_k}$ donc :

$$x_{p+1} = \prod_{k=0}^p y_k^{b_k} = \prod_{k=0}^p a^{(b_k 2^k)} = a^{\sum_{k=0}^p b_k 2^k} = a^n.$$

Il s'agit donc d'un algorithme de calcul de a^n , dont le coût est un $O(p) = O(\log n)$. On l'appelle l'*algorithme d'exponentiation rapide* ; on connaît surtout sa version récursive :

```
let rec exp a = fonction
  0 -> 1
  | n when n mod 2 = 0 -> exp (a * a) (n / 2)
  | n -> a * exp (a * a) (n / 2) ;;
```

3. Exercices

1 – Déterminer l'itérateur associé à chacune des deux suites $(f_n)_{n \in \mathbb{N}}$ et $(g_n)_{n \in \mathbb{N}}$ définies par :

$$\begin{cases} f_0 = f_1 = 1 \\ f_{n+2} = f_{n+1} + f_n \end{cases} \quad \text{et} \quad \begin{cases} g_0 = g_1 = 1 \\ g_{n+2} = g_{n+1} + (-1)^n g_n \end{cases}$$

Rédiger des versions itératives et récursives des fonctions permettant le calcul de f_n et de g_n .

2 – Démontrer que l'algorithme suivant de multiplication des entiers est correct :

```
let multiplie a b =
  let x = ref 0 and y = ref a and z = ref b in
  while !z > 0 do x := !x + !y * (!z mod 2) ;
    y := 2 * !y ;
    z := !z / 2
  done ;
  !x ;;
```

Analyser son coût, et en donner une version récursive.

3 – Déterminer ce que calcule la fonction suivante, et prouvez-le :

```
let f u =  
  let x = ref 0 and y = ref u in  
  let c = ref 1 and d = ref 1 in  
  while (!y > 0) or (!c > 0) do  
    let a = !y mod 2 in  
    if (a + !c) mod 2 = 1 then x := !x + !d ;  
    c := a * !c ;  
    d := 2 * !d ;  
    y := !y / 2 done ;  
  !x ;;
```

■