Option informatique, cours 5

Programmation impérative en OCaml et types de données mutables

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

Généralités sur le style impératif

Exemple de la boucle while

En python, les boucles **for** et **while** permettent de réaliser des opérations en séquence : les unes à la suite des autres. En voici un exemple :

```
i = 5
while (i>0):
        print i
        i = i-1
```

Cette construction suppose

1. que la condition puisse, au fil des itérations, passer d'une valeur vrai à faux afin que la boucle s'arrête. La valeur de la variable i doit donc pouvoir **évoluer**.
2. que les instruction contenues dans la boucle modifient **l'état** de l'environnement pour que la boucle ait un sens. Sinon, il suffirait d'évaluer la dernière expression.

Exemple de la boucle while

Ce style de programmation s'appelle **programmation impérative**. Les instructions exécutées en séquence modifient l'état de l'environnement pour aboutir au résultat souhaité.

Il s'oppose au style **fonctionnel**, vu jusqu'à présent, et dans lequel ces séquences étaient inutiles en raison de l'absence d'effet de bord.

Exemple de la boucle while

Notre boucle en python est donc écrite dans un style **impératif**.

```
i = 5
while (i>0):
        print i
        i = i-1
```

Un équivalent dans le style **fonctionnel** en OCaml utiliserait la récursivité :

```
let rec f = function
        | 0 -> ()
        | k -> print_int k; f (k-1)
in f 5;;
```

À noter que cette dernière fonction utilise déjà l'opérateur séquence ; en raison de l'effet de bord de l'affichage.

Exemple de la boucle while

Le style **impératif** nécessite donc :

- d'avoir la possibilité d'exécuter des instructions en séquence (on dispose pour cela de l'opérateur ; en Ocaml)
- que ses instructions puissent avoir un effet de bord, comme la modification d'une structure de donnée mutable par exemple (on a déjà vu les fonction du type `print_int`).

Bien que ce ne soit pas le style habituel, il est possible de concevoir des algorithmes en OCaml en style impératif, et donc de modifier des variables, d'utiliser des structures itératives, comme en python.

Types mutables en OCaml

Champs mutables d'un enregistrement

On rappelle la syntaxe pour définir un type enregistrement :

```
type complexe = {re: float; im: float};;
type entier = {k: int};;
```

On accède aux champs à l'aide de l'opérateur .

```
let i = {re=0.; im=1.;};;
i.im;;
let compteur = {k=5};;
compteur.k;;
```

Dans ce dernier exemple, le type entier n'est qu'un entier **encapsulé** dans un enregistrement, ce qui n'a pas grand intérêt à ce stade, à part alourdir la syntaxe...

Champs mutables d'un enregistrement

Pour qu'un champ d'un enregistrement puisse être modifié (ont dit qu'il est mutable), il suffit d'ajouter le mot clé `mutable` au moment de la définition de l'enregistrement.

```
type entier = {mutable k: int};;
```

Ainsi, il va être possible de modifier ce champ et l'encapsulation prend tout son intérêt, même si elle reste lourde sur le plan de la syntaxe.

Champs mutables d'un enregistrement

La séquence d'instructions suivantes permet de modifier le champ mutable k du type entier :

```
type entier = {mutable k: int};;
let compteur = {k=1};;
compteur.k <- 10;;
```

Cette dernière instruction utilise l'opérateur d'affectation <-. Elle est de type `unit` car elle agit uniquement par effet de bord.

En réalité, cette possibilité de créer des types mutables est peu utilisée car lourde. Il existe des types mutables prédéfinis : le types polymorphes `'a ref` et `array`.

Le type 'a ref

Ocaml comporte un type bien utile lorsqu'il s'agit de créer une variable mutable : le type `'a ref`, qui pourrait être défini de la façon suivante (il n'est pas nécessaire de le faire car ce type est déjà défini) :

```
type 'a ref = {mutable contents: 'a};;
```

La fonction (prédéfinie) `ref` joue le rôle de constructeur. Elle est polymorphe et du type `'a -> 'a ref`.

```
let compteur = ref 5;;
```

Cette dernière instruction créée une référence à la valeur 5. compteur est alors du type `int ref`.

Le type 'a ref

Pour modifier une référence, il faut utiliser l'opérateur d'**assignation** :=.

```
let compteur = ref(5);;
compteur := 10;;
```

Pour accéder à la valeur courante d'une référence, il faut utiliser l'opérateur préfixe de **déréférencement** !.

```
!compteur;;
compteur := !compteur +1 ;;
```

C'est une première manière de travailler avec des données mutables en Ocaml.

Le type array : introduction

Les tableaux correspondent, en langage OCaml, aux listes de Python au sens où ce sont des structures **mutables**. Les listes d'OCaml sont non mutables, mais ce n'est pas la seule différence.

Contrairement aux listes d'OCaml, on peut accéder à n'importe quel élément d'un tableau en temps **constant**. Avec une liste, il faut parcourir récursivement la liste de sorte que la complexité de l'accès au n-ième élément est **linéaire**.

En contrepartie, la taille d'un tableau est fixée au moment de sa création. L'ajout d'un élément se fait donc en le recopiant (complexité **linéaire**), alors que l'ajout d'un élément en tête de liste se fait en temps **constant**.

Le type array : introduction

En résumé, outre le caractère mutable ou non mutable, la complexité des opérations que l'on peut faire sur une structure de données (list ou array) peut varier en fonction du langage ou de l'implémentation.

Les caractéristiques importantes associées à une structure de données sont :

- la façon de représenter des données (arbre, liste)
- le fait d'être mutable ou non mutable
- les opérations (accès à un élément, ...) associées à leur **complexité**.

On choisira, pour résoudre un problème donné, la structure de données la plus adaptée.

Le type array

La création d'un tableau, l'accès et la modification d'un élément se font selon la syntaxe suivante :

```
let tableau = [| 1;2;3 |];;
tableau.(0) <- (tableau.(0)-1);;
```

Notez que la syntaxe se rapproche plus de celle des champs mutables d'un type enregistrement (avec les opérateurs `<-` et `.`) que de celle des références.

La tentative d'accéder à un index supérieur à la taille du tableau lève une exception `Invalid_argument` avec le paramètre `"index out of bounds"`

Le module Array

Dans le langage Ocaml, des fonctions existent pour manipuler des tableaux. Elles appartiennent au module Array. On peut donc les appeler :

- soit en préfixant la fonction par `Array.`
- soit en important toutes les fonctions de ce module à l'aide de la commande `open Array;;`

En particulier, les fonctions `get` et `set`, respectivement de type `'a array -> int -> 'a` et `'a array -> int -> 'a -> unit`, permettent (au prix d'une syntaxe plus lourde) de se passer des opérateurs `.()` et `<-`.

```
open Array;;
let tableau = [| 1;2;3 |];;
set tableau 0 ((get tableau 0) -1);;
```

Création d'un tableau

Attention, si les valeurs contenues dans un tableau sont modifiables, le tableau lui même (et en particulier sa longueur) est fixé lors de la création. La longueur est accessible à l'aide de la fonction `Array.length`, du type `'a array -> int`.

Si l'on souhaite créer un tableau sans en énumérer les éléments (par exemple lorsque la taille est inconnue lors de l'écriture du programme), on utilise la fonction `Array.make`, du type `int-> 'a -> 'a array`. Le premier argument est la taille souhaitée et le deuxième argument est la valeur qui sera répliquée dans tout le tableau.

Modification d'un tableau par effet de bord

On rappelle qu'un tableau est un type de données **mutable**.

```
let tableau1 = [| 1;2;3 |];;
let tableau2 = tableau1;;
tableau1.(0) <- tableau1.(0)-1;;
```

En particulier, dans cette séquence d'instructions, les **deux** tableaux ont été modifiés. Ceci est particulièrement important lorsqu'on écrit une fonction qui prend un tableau en argument, et le modifie dans le corps de la fonction.

Tableaux bidimentionnels

On peut créer un tableau bidimensionnel (une matrice) à l'aide de la fonction `make_matrix`, qui est de type `int->int->'a->'a array array`. Les deux premiers arguments correspondent à la taille et le troisième contient la valeur qui sera répliquée dans le tableau nouvellement créé.

Une matrice n'est rien d'autre qu'un tableau de tableaux. L'élément d'indice (i, j) de la matrice m est accessible à l'aide de l'expression `m.(i).(j)`. L'opérateur d'assignation est le même que pour les tableaux unidimensionnels.

Structures itératives

En style impératif, on souhaite exécuter plusieurs instructions à la suite. On peut pour cela utiliser plusieurs évaluations successives par l'interpréteur :

```
let tableau = Array.make 3 0;;
tableau.(0) <- 1;;
tableau.(1) <- 2;;
tableau.(2) <- 3;;
```

C'est une première forme, certes rudimentaire, d'itération.

En style impératif, on peut également utiliser l'opérateur ;, appelé séquence.

```
let tt = Array.make 3 0;;
tt.(0) <- 1; tt.(1) <- 2; tt.(2) <- 3;;
```

Cette écriture n'est toujours pas très commode si on souhaite exécuter beaucoup d'instructions.

Structures itératives en OCaml

La syntaxe d'une boucle `for` est la suivante :

```
let tt = Array.make 3 0;;
for k=0 to (Array.length tt)-1 do
        tt.(k) <- k+1
done;;
```

Tout comme en python (et d'autres langages) le compteur de boucle n'a pas besoin d'être déclaré avant la boucle, ni incrémenté explicitement dans le corps de la boucle.

Toujours comme en python, la boucle `for` est une structure itérative adaptée au cas où l'on connait, au moment d'exécuter la première itération, le nombre de fois que le bloc d'instructions devra être répété.

Structures itératives en OCaml

La syntaxe d'une boucle `while` est la suivante :

```
let tt = Array.make 3 0;;
let k = ref 0;;
while !k<3 do
        tt.(!k) <- !k+1;
        k := !k+1
done;;
```

L'utilisation d'une boucle `while` nécessite que la condition devienne fausse au bout d'un certain nombre (fini) d'itérations : elle doit donc faire intervenir une structure mutable.

Dans l'exemple ci-dessus, la variable mutable `k` est de type `int ref` et a été définie juste avant.

De plus, les instructions du corps de la boucle doivent nécessairement modifier la condition (sauf si on rattrape une exception).

```
let tt = Array.make 3 0;;
let k = ref 0;;
try while true do
        tt.(!k) <- !k+1;
        k := !k+1
done with | Invalid_argument _ ->();;
```

Applications

Application 1 : palindromes

Une première application naturelle de la structure de tableau utilise le fait que l'accès à n'importe quel élément se fait en temps constant.

Définir une fonction `palindrome` de type `'a array -> bool` et de complexité linéaire, qui renvoie vrai si le tableau passé en argument est un palindrome et faux sinon.

On remarque qu'il est aussi possible de définir une telle fonction prenant en argument une liste et que la complexité est aussi linéaire :

```
let palindrome l =
  l = List.rev l;;
```

Application 1 : palindromes

Propositions de correction :

```
let palindrome t =
  let n =Array.length t and res=ref true in
  for k=0 to n/2 do
      if t.(k) != t.(n-1-k) then res :=false
  done;
  !res;;
```

```
let palindrome t =
  let n =Array.length t and  k=ref 0 in
  while !k<= n/2 && (t.(!k) = t.(n-1-(!k))) do
      k := !k+1
  done;
  (!k)>(n/2) ;;
```

Application 2 : Coefficients binomiaux et programmation dynamique

Certaine fonctions récursives font intervenir plusieurs appels récursifs. C'est le cas, par exemple du calcul du n-ième terme de la suite de Fibonacci ou du coefficient binomial :

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

let rec binom n = function
  | 0 -> 1
  | k when k > n -> 0
  | k -> binom (n-1) (k-1) + binom (n-1) k;;
```

Application 2 : Coefficients binomiaux et programmation dynamique

Ces fonctions ont plusieurs points en commun :

- faire intervenir deux appels récursifs
- être d'une complexité exponentielle
- plusieurs appels récursifs avec les mêmes arguments sont nécessaires, ce qui est dommage.

Une autre stratégie est possible : la **programmation dynamique**. Comme on le verra, elle permet de ne calculer qu'une fois chacun des résultats intermédiaires qui permettent d'obtenir le résultat final. En conséquence, la complexité est améliorée.

Application 2 : Coefficients binomiaux et programmation dynamique

Pour la suite de Fibonacci, on peut stocker dans un tableau tous les appels `fibo k` nécessaires.

```
let fibo2 n =
  let m = Array.make n 1 in
  for k=2 to (n-1) do m.(k)<-m.(k-1)+m.(k-2) done;
  m.(n-1);;
```

On retrouve une implémentation itérative, dont la complexité est **linéaire** mais la création d'un tableau n'est pas nécessaire :

```
let fibo3 n =
  let rec aux acc a b = match acc with
    |0 -> a
    |n -> aux (k-1) b (a+b)
  in aux n 0 1;;
```

Application 2 : Coefficients binomiaux et programmation dynamique

Pour le calcul des coefficients binomiaux, on peut utiliser une matrice pour stocker les résultats intermédiaires.

C'est le principe de la **programmation dynamique** : on stocke dans une structure de données les résultats des appels récursifs nécessaires et on fait en sorte de ne calculer chaque combinaison qu'une seule fois.

Application 2 : Coefficients binomiaux et programmation dynamique

```
let binomi k n =
  if k>n then 0 else
  let m=Array.make_matrix (n+1) (k+1) 0 in
  for i=0 to n do
    m.(i).(0) <-1;
    for j=1 to (min i k)do
      m.(i).(j) <- m.(i-1).(j)+m.(i-1).(j-1);
    done;
  done;
  m.(n).(k);;
```

Application 3 : crible d'Eratosthène

Pour trouver l'ensemble des nombres premiers plus petits qu'un entier n fixé, on peut procéder de la façon suivante :

- on construit un tableau représentant les entiers de 2 à n,
- pour chaque entier, soit il n'est pas premier et on passe au suivant, soit il est premier et, dans ce cas, on le conserve et l'on "barre" tous ses multiples.

Implémentez une fonction `eratosthene` de type `int -> int list`, qui prend en argument un entier et renvoie la liste des nombres premiers inférieurs ou égaux à n.

Application 3 : crible d'Eratosthène

Proposition de correction (boucle for):

```
let eratosthene n =
  let t= Array.make(n-1) 1 in
  let rec aux = function
    |k when k=(n+1) -> []
    |k when t.(k-2)=0 -> aux(k+1)
    |k -> for q=2 to (n/k) do
            t.(q*k-2) <- 0
          done;
          k::(aux (k+1))
  in aux 2;;
```

Application 3 : crible d'Eratosthène

Proposition de correction (boucle for + terminal):

```
let eratosthene n =
  let t= Array.make(n-1) 1 in
  let rec aux acc = function
    |k when k>(n-2) -> acc
    |k when t.(k)=0 -> aux acc (k+1)
    |k -> for q=2 to (n/(k+2)) do
            t.(q*(k+2)-2) <- 0
          done;
          aux ((k+2)::acc) (k+1)
  in List.rev (aux [] 0);;
```

Application 3 : crible d'Eratosthène

Proposition de correction (boucle while):

```
let eratosthene n =
  let t= Array.make(n-1) 1 in
  let rec aux = function
    |k when k=(n+1) -> []
    |k when t.(k-2)=0 -> aux(k+1)
    |k -> let i = ref(2*k) in
            while !i<=n do
                t.(!i-2) <- 0; i := !i+k
            done;
          k::(aux (k+1))
  in aux 2;;
```

Application 3 : crible d'Eratosthène

Proposition de correction (boucle while + terminal):

```
let eratosthene n =
  let t= Array.make(n-1) 1 in
  let rec aux acc = function
    |k when k>(n-2) -> acc
    |k when t.(k)=0 -> aux acc (k+1)
    |k -> let i = ref( 2*(k+2) ) in
          while (!i-2)<= n-2 do
            t.(!i-2) <- 0; i:= !i+(k+2)
          done;
          aux ((k+2)::acc) (k+1)
  in List.rev (aux [] 0);;
```

Cette dernière solution est moins élégante car on doit définir une référence pour servir de compteur de boucle et la mettre à jour à chaque itération.