V - Programmation impérative

30 janvier 2024

1. Effets de bord, variables, types mutables

1.1. Effets de bord

La programmation impérative repose sur un principe différent de la programmation fonctionnelle que nous avons utilisé jusqu'à présent. Suivant ce paradigme, un programme est conçu comme une séquence d'instructions ayant pour effet de modifier l'état de l'ordinateur. Le processeur peut agir sur l'état des périphériques d'entrée/sortie et sur l'état de la mémoire.

On dira qu'une fonction est dite *à effet de bord* lorsqu'elle modifie un état en dehors de son environnement local. Par extension, un opérateur est dit *à effet de bord* lorsqu'il modifie l'un de ses opérande. Par exemple, en python, modifier la valeur d'une variable est un effet de bord.

Il est fréquent qu'une fonction à effet de bord n'ait pas besoin de renvoyer une valeur. Dans ce cas, elle renverra une valeur de type `unit`. La seule valeur de type `unit` est `()`

Remarque : Il s'agit de l'équivalent de `None` en Python (dont le type est `NoneType`).

Par exemple, les fonctions d'affichage sont des fonctions à effet de bord : `print_string`, `print_char`, `print_int`, `print_float`, `print_newline`, `print_endline`.

Une fonction qui n'a pas besoin d'un argument aura un argument de type `unit`, on la définira et on l'appellera donc avec `()`

```
[1] : let f () = print_endline "Coucou" ; ;
```

```
[1] : val f  : unit -> unit = <fun>
```

La fonction `print_endline : string -> unit` permet d'afficher une chaîne de caractères et de revenir ensuite à la ligne.

```
[2] : f () ; ;
```

```
Coucou
```

```
[2] : -  : unit = ()
```

1.2. Séquences d'instructions

En Ocaml, les séquences d'instructions sont des expressions séparées par des points-virgules. L'évaluation de

$$e_1 \ ; \ e_2 \ ; \ ... \ ; \ e_n$$

provoque les effets éventuels de ces n expressions dans l'ordre, et a la valeur de la dernière expression.

Ce dernier point implique qu'il n'y a aucun intérêt à utiliser une séquence si les expressions dont la valeur n'est pas utilisée (c'est-à-dire toutes sauf la dernière) n'ont pas d'effet de bord.

Par ailleurs, ces différentes expressions (sauf éventuellement la dernière) sont la plupart du temps de type `unit` puisque seule la valeur de la dernière expression est prise en compte.

[3] :
```
let classe nom numero =
  print_string "Vivent les " ;
  print_string nom ;
  print_int numero ;
  print_newline () ;
  numero * numero ; ;
```

[3] :
```
val classe  : string -> int -> int = <fun>
```

[4] :
```
classe "MPSI" 3 ; ;
```

```
Vivent les MPSI3
```

[4] :
```
-  : int = 9
```

Lorsqu'on souhaite utiliser une séquence à la place d'une expression dans une instruction conditionnelle, il est nécessaire d'encadrer celle-ci par les mots-clés `begin` et `end` ou d'utiliser des parenthèses.

[5] :
```
let x = 0
in if x = 0 then (print_endline "x est nuuuuul"  ; print_endline "Vive␣
 ↪l'info  !") ; ;
```

```
x est nuuuuul
Vive l'info  !
```

[5] :
```
-  : unit = ()
```

[6] :
```
let est_pair n =
  if n mod 2 = 0
  then
    (print_endline "nombre pair" ;true)
  else
    begin
      print_endline "nombre impair" ;
      false
    end
 ; ;

est_pair 3 ; ;
est_pair 16 ; ;
```

```
nombre impair
nombre pair
```

```
[6] : val est_pair  : int -> bool = <fun>
      -  : bool = false
      -  : bool = true
```

1.3. Enregistrements avec champ modifiable

Dans un type enregistrement, on peut déclarer un ou plusieurs champs comme modifiable (ou *mutable*).

```
[7] : type etudiant = {
        nom  : string ;
        prenom  : string ;
        mutable classe  : string ;
      } ; ;

      let e = {
        nom = "Tournesol" ;
        prenom = "Tryphon" ;
        classe = "MPSI"
      } ; ;
```

```
[7] : type etudiant = { nom  : string ; prenom  : string ; mutable classe  : string ;
       ↪ }
      val e  : etudiant = {nom = "Tournesol" ; prenom = "Tryphon" ; classe = "MPSI"}
```

Si le champ `c` de l'enregistrement `a` est mutable, la modification de sa valeur se fait avec `a.c <-` e où e est une expression.

```
[8] : e.classe <- "MP" ; ;
```

```
[8] : -  : unit = ()
```

```
[9] : () ; ;
```

```
[9] : -  : unit = ()
```

Il n'y a pas d'instruction en OCaml. `e.classe <- "MP"` est en réalité une expression dont la valeur est `()` et dont le type est `unit`.

1.4. Références

a. Utilisation de variables en OCaml

Les variables en OCaml rencontrées jusqu'à présent ne sont pas des variables au sens traditionnel des langages de programmation, puisqu'il est impossible de modifier leur valeur. En programmation impérative, il est indispensable de pouvoir utiliser des variables modifiables pour mémoriser une information évoluant au fil du programme.

En OCaml, on utilise alors une *référence* vers une valeur, c'est-à-dire une case mémoire dont on peut lire et écrire le contenu.

En pratique, on définit une référence avec le mot-clé `ref` (qui est en fait une fonction `'a -> 'a ref`).

```
[10] : let x = ref 0 ; ;
       let bidule = ref "coucou" ; ;

[10] : val x  : int ref = {contents = 0}
       val bidule  : string ref = {contents = "coucou"}
```

On remarque que son type correspond à un type enregistrement `'a ref` qui contient un unique champ de type `'a` appelé `contents`.

Le type de la valeur pointée par une référence est fixé à la création. Une référence pointant vers un objet de type `'a` a le type `'a ref`.

Pour accéder à la valeur de la référence `x`, on utilise l'*opérateur de référencement* `!` suivi du nom de la référence.

Pour modifier une référence, on utilise l'*opérateur d'affectation* `:=` précédé du nom de la référence, et suivi d'une expression.

```
[11] : x  := 3 ; ;

[11] : -  : unit = ()

[12] : x  :=  !x + 1 ; ;

[12] : -  : unit = ()

[13] :  !x ; ;

[13] : -  : int = 4
```

L'opérateur d'affectation `:=` est à effet de bord.

b. Différence entre une définition et une référence

Une définition établit une liaison permanente entre un nom et une valeur, alors qu'une référence établit une liaison entre un nom et un emplacement en mémoire. Il importe donc de bien différencier `let s = 0` (liaison entre le nom `s` et la valeur `0`) et `let s = ref 0` (liaison entre le nom `s` et un emplacement en mémoire contenant la valeur `0`). Une autre différence est le comportement statique d'une définition, par opposition au comportement dynamique d'une référence, comme l'illustre les deux séquences d'instructions suivantes ci-dessous :

Tout d'abord :

```
[14] : let a = 1  ; ;
       let incr x = x + a  ; ;
       let a = 2  ; ;
       incr 0  ; ;

[14] : val a  : int = 1
       val incr  : int -> int = <fun>
       val a  : int = 2
       -  : int = 1
```

Et ensuite :

[15] :
```
let a = ref 1 ;;
let incr x = x + !a ;;a := 2 ;;
incr 0 ;;
```

[15] :
```
val a : int ref = {contents = 1}
val incr : int -> int = <fun>
- : unit = ()
- : int = 2
```

c. Égalités physique et structurelle

Par conséquent il existe en OCaml deux types d'égalité : l'une qui teste l'égalité entre deux valeurs, l'autre qui détermine si deux objets occupent le même emplacement mémoire. La première est appelée *égalité structurelle* et est représentée par le symbole `=`. La seconde est appelée *égalité physique* et est représentée par le symbole `==`.

[16] :
```
let x = ref 0 ;;
let y = ref 0 ;;
!x = !y ;;
!x == !y ;;
x = y ;;
x == y ;;
```

[16] :
```
val x : int ref = {contents = 0}
val y : int ref = {contents = 0}
- : bool = true
- : bool = true
- : bool = true
- : bool = false
```

2. Boucles

2.1. Boucles inconditionnelles

Il existe deux types de boucles inconditionnelles en OCaml :

— `for` *indice* `=` e_1 `to` e_2 `do` e_3 `done`

— `for` *indice* `=` e_1 `downto` e_2 `do` e_3 `done`

Dans le premier cas, l'indice de boucle (de type `int`) est incrémenté de 1 en 1 entre les valeurs des expressions e_1 et e_2, en effectuant le calcul de e_3 à chaque fois (il est donc préférable que e_3 ait un effet de bord, et pertinent que e_3 soit de type `unit`).

Le second cas est identique, mais l'indice de boucle est décrémenté.

[17] :
```
for i = 1 to 10 do print_int i ; print_char ' ' done ;
print_newline () ;;
```

```
1 2 3 4 5 6 7 8 9 10
```

[17] :
```
- : unit = ()
```

```
[18] : for i = 10 downto 1 do print_int i ;  print_char ' ' done ;
       print_newline () ; ;
```

```
10 9 8 7 6 5 4 3 2 1
```

```
[18] : -  : unit = ()
```

⚠ Les deux bornes sont atteintes.

2.2. Boucles conditionnelles

La syntaxe d'une boucle conditionnelle est la suivante :

— `while` e_1 `do` e_2 `done`

Tant que l'expression e_1 (nécessairement de type `bool`) a la valeur `true`, on évalue l'expression e_2.

```
[19] : let a = ref 2024 in
       while !a > 0 do
         print_int (!a mod 2) ;
         a := !a / 2
       done ;
       print_newline () ; ;
```

```
00010111111
```

```
[19] : -  : unit = ()
```

On lira bien évidemment l'affichage de droite à gauche.

Remarque : Si l'expression e_2 n'a pas d'effet de bord, il est peu probable que la condition puisse cesser d'être vérifiée...

3. Structure de tableaux

Les *tableaux* (ou vecteurs) sont des structures correspondant aux vecteurs mathématiques. Un tableau est une suite finie de valeurs **de même type**, modifiables, l'accès à une composante se faisant à temps constant.

La *longueur* d'un tableau (son nombre d'éléments) est fixée lors de la création et **ne peut être modifiée**

Un tableau est délimité par `[|` et `|]`, ses éléments sont séparés par des points-virgules.

Un tableau contenant des valeurs de type `'a` a pour type `'a array`.

```
[20] : let t1 = [|1 ;2 ;3 ;4 ;5 ;6|] ; ;
```

```
[20] : val t1  : int array = [|1 ; 2 ; 3 ; 4 ; 5 ; 6|]
```

La fonction `Array.length` renvoie la longueur du tableau passé en argument.

```
[21] : Array.length t1 ; ;
[21] : -  : int = 6
```

⚠ En OCaml, les éléments d'un tableau de longueur n sont numérotés de 0 à $n-1$.

L'accès à l'élément d'indice `i` du tableau `t` s'écrit : `t.(i)`.

Cet élément peut être modifié par `t.(i) <-` e où e est une expression.

```
[22] : t1.(5) ; ;
[22] : -  : int = 6
[23] : t1.(2) <- 42 ; ;
[23] : -  : unit = ()
[24] : t1 ; ;
[24] : -  : int array = [|1 ; 2 ; 42 ; 4 ; 5 ; 6|]
[25] : t1.(6) ; ;
[25] : Exception : Invalid_argument "index out of bounds".
[26] : t1.(-1) ; ;
[26] : Exception : Invalid_argument "index out of bounds".
```

Pour créer un tableau, on peut utiliser la fonction `Array.make` : `Array.make n x` renvoie un tableau de taille `n` dans lequel tous les éléments valent `x`.

```
[27] : Array.make 3 'a' ; ;
[27] : -  : char array = [|'a' ; 'a' ; 'a'|]
```

⚠ Les éléments du tableaux sont alors physiquement tous égaux.

```
[28] : let tabtab = Array.make 3 [|1 ;1|] ; ;
[28] : val tabtab  : int array array = [|[|1 ; 1|] ; [|1 ; 1|] ; [|1 ; 1|]|]
[29] : tabtab.(0).(0) <- 1515 ; ;
[29] : -  : unit = ()
[30] : tabtab ; ;
[30] : -  : int array array = [|[|1515 ; 1|] ; [|1515 ; 1|] ; [|1515 ; 1|]|]
```

Pour initialiser une matrice, on utilisera donc la fonction `Array.make_matrix : int -> int -> 'a -> 'a array array`

```
[31] : let tabtab = Array.make_matrix 2 3 0 ; ;
```

```
[31] : val tabtab  : int array array = [|[|0 ; 0 ; 0|] ; [|0 ; 0 ; 0|]|]
```

```
[32] : tabtab.(0).(0) <- 1515 ; ;
       tabtab ; ;
```

```
[32] : -  : unit = ()
       -  : int array array = [|[|1515 ; 0 ; 0|] ; [|0 ; 0 ; 0|]|]
```

Enfin, il est possible de créer un tableau avec la fonction `Array.init  : int -> (int -> 'a) -> 'a array` en précisant la longueur du tableau une fonction calculant l'élément d'indice i en fonction de i.

```
[33] : let tab = Array.init 5 (fun i -> 2*i) ; ;
```

```
[33] : val tab  : int array = [|0 ; 2 ; 4 ; 6 ; 8|]
```

On notera bien les différences entre listes et tableaux : on accède au premier élément d'une liste en temps constant, et on modifie sa longueur en temps constant, par contre accèder aux autres éléments demande de parcourir d'abord tous les éléments précédents, ce qui prend du temps. À l'inverse, on accède à et on modifie tous les éléments d'un tableau en temps constant, mais changer la longueur d'un tableau nécessite de le recopier dans un nouveau tableau ayant la taille choisie, ce qui est de complexité linéaire en la longueur du tableau.

4. Exercices divers

4.1. Exercice 1

Prévoir la réponse de l'interpréteur de commandes :

```
[ ] : let x = ref 0 ; ;
      let z = x ; ;
      x  := 4 ; ;
```

```
[ ] : !z ; ;
```

4.2. Exercice 2

Prévoir la réponse de l'interpréteur de commandes :

```
[ ] : let a = ref 1 ; ;
      let f x = x + !a ; ;
      f 3 ; ;
```

```
[ ] : a  := 4 ; ;
      f 3 ; ;
```

4.3. Exercice 3

— Écrire une fonction `copy : 'a array -> 'a array` qui prend en argument un tableau et en renvoie une copie. Quelle est sa complexité ?

— Écrire une fonction `mem : 'a -> 'a array -> bool` testant l'appartenance d'un élément à un tableau (on pourra l'écrire en impératif et/ou en récursif). Quelle est sa complexité ?

— Écrire une fonction `map : ('a -> 'b) -> 'a array -> 'b array` qui prend en argument une fonction `f` de type `'a -> 'b` et un tableau `[|a1 ; ... ; an|]` d'éléments de type `'a` et qui renvoie le tableau `[|f a1 ; ... ; f an|]`.

— Écrire une fonction `exists : ('a -> bool) -> 'a array -> bool` qui prend en argument une fonction `f` de type `'a -> bool` et un tableau `[|a1 ; ... ; an|]` d'éléments de type `'a` et qui renvoie `true` si au moins un élément du tableau vérifie `f`, `false` sinon (on pourra l'écrire en impératif et/ou en récursif).

— Écrire une fonction `for_all : ('a -> bool) -> 'a array -> bool` qui prend en argument une fonction `f` de type `'a -> bool` et un tableau `[|a1 ; ... ; an|]` d'éléments de type `'a` et qui renvoie `true` si tous les éléments du tableau vérifie `f`, `false` sinon (on pourra l'écrire en impératif et/ou en récursif).

Ces fonctions sont déjà implémentées dans le module `Array`.