cours4

November 21, 2023

1 Programmation fonctionnelle

1.1 Semaine 4 : Programmer en style impératif

Etienne Lozes - Université Nice Sophia Antipolis - 2019

---

[1]:
```
let () = ()
```

```
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,…";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads
```

1.2 Mutation et état

Pour le moment, les variables et les structures de données que nous avons rencontrées sont *non mutables* (listes, chaînes de caractères, arbres, etc): une fois qu'une variable est définie, on ne peut pas changer sa valeur - au mieux on peut la masquer par une nouvelle définition de variable portant le même nom.

Dans les langages impératifs, en revanche, la valeur d'une variable change en permanence, de même que le contenu des structures de données en mémoire, les informations systèmes (fichiers ouverts, communications établies, etc).

D'une manière générale, **un programme impératif modifie l'état** dans lequel le programme se trouve à chaque instruction qu'il exécute.

Certains langages fonctionnels, notamment Haskell, sont farouchement déclaratifs: rien de ce qui a été défini ne peut être modifié, **l'état n'existe pas**. C'est un peu extrémiste, mais cela permet de faire en permanence de l'**évaluation paresseuse**, une idée très intéressante dont on aura l'occasion de reparler…

OCaml est un langage un peu plus consensuel. Il nous pousse certes à programmer sans modifier l'état, mais il nous laisse aussi la possibilité de programmer de façon impérative, d'une façon très similaire à ce qu'on ferait en Python, C, Java, etc.

1.3 Les tableaux

Les tableaux (*arrays* en anglais) sont une structure de données mutable fréquemment utilisée en OCaml. Pour créer un tableau, on peut lister les valeurs qu'il contient. La syntaxe est presque la même que pour les listes, sauf qu'on écrit `[|...|]` au lieu de `[...]`.

[2]:
```
let tab = [|1;2;3|]   (* <==> tab = [1,2,3] en Python *)
```

[2]:
```
val tab : int array = [|1; 2; 3|]
```

`Array.make n v` renvoie un nouveau tableau de `n` cases contenant toutes la valeur `v`.

[3]:
```
let tab2 = Array.make 50 '#' (* un tableau de 50 caractères '#'*)
```

[3]:
```
val tab2 : char array =
  [|'#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#';
    '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#';
    '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#';
    '#'; '#'; '#'; '#'; '#'; '#'; '#'; '#'|]
```

`Array.init n f` renvoie le tableau `[|f(0); f(1);...;f(n-1)|]`.

On utilise souvent `Array.init` avec une **fonction anonyme** de la forme `fun x -> ...`

[4]:
```
Array.init 10 (fun i -> i * i)    (* <=> [i*i for i in range(10)] en Python *)
```

[4]:
```
- : int array = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

Contrairement aux listes, l'accès au $i$ème élément d'un tableau se fait en temps constant.

[5]:
```
tab.(1)   (* <==> tab[1] en Python *)
```

[5]:
```
- : int = 2
```

[6]:
```
tab.(-1)
```

```
Exception: Invalid_argument "index out of bounds".
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Attention, on n'est pas en Python, les seuls indices valables sont ceux compris entre 0 et `(Array.length tab) - 1` !

Voyons donc maintenant comment faire muter un tableau.

[7]:
```
tab.(0) <- 0    (* mutation! *)
```

[7]:
```
- : unit = ()
```

[8]:
```
tab  (* le contenu de tab a changé de manière irréversible. *)
```

[8]:
```
- : int array = [|0; 2; 3|]
```

Notez le type `unit` de l'expression `tab.(0) <- 0`: il s'agit d'une instruction, elle n'a pas de valeur (ou plutôt, elle a la valeur `()`), mais elle fait un **effet de bord**.

1.4 Les enregistrements mutables

Par défaut, les champs des enregistrements ne sont pas mutables. On peut cependant rendre mutable un champs avec le mot-clé `mutable`.

[9]:
```
type etat_civil = {
  date_naissance: int;           (* champs non mutable *)
  mutable date_deces: int option (* champs mutable *)
}
```

[9]:
```
type etat_civil = { date_naissance : int; mutable date_deces : int option; }
```

[10]:
```
let christ = {date_naissance= -33; date_deces = Some(0)}
```

[10]:
```
val christ : etat_civil = {date_naissance = -33; date_deces = Some 0}
```

[11]:
```
christ.date_deces <- None
```

[11]:
```
- : unit = ()
```

[12]:
```
christ
```

[12]:
```
- : etat_civil = {date_naissance = -33; date_deces = None}
```

Si le champs n'a pas été déclaré mutable, pas le droit d'y toucher

[13]:
```
christ.date_naissance <- -34
```

```
File "[13]", line 1, characters 0-28:
1 | christ.date_naissance <- -34
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The record field date_naissance is not mutable
```

1.5 Les références

Une référence est une case mémoire qui contient une valeur, autrement dit c'est une sorte de tableau de dimension 1.

Pour OCaml, c'est simplement un enregistrement à un seul champs nommé `contents` qui est mutable. La librairie standard définit le type `'a ref` ainsi

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

Pour créer une nouvelle référence, on utilise la fonction `ref`

[14]:
```
let r = ref 42  (* création d'une nouvelle référence qui contient 42 *)
(* <=> let r = Array.make 1 42 *)
(* <=> let r = {content=42} *)
```

[14]:
```
val r : int ref = {contents = 42}
```

La valeur de `r` n'est pas 42. La valeur de `r` est "une certaine adresse mémoire". Cela n'a même pas de sens de comparer l'entier 42 à la référence r.

[15]:
```
if r = 42 then print_string "r = 42" else print_string "r <> 42"
```

```
File "[15]", line 1, characters 7-9:
1 | if r = 42 then print_string "r = 42" else print_string "r <> 42"
           ^^
Error: This expression has type int but an expression was expected of type
         int ref
```

À chaque appel à `ref`, une nouvelle référence est créée, différente des précédentes. On parle d'**allocation mémoire** (`malloc` en C, `new` en Java, souvent implicite en Python)

[16]:
```
let r2 = ref 42
```

[16]:
```
val r2 : int ref = {contents = 42}
```

[17]:
```
r == r2    (* r et r2 sont deux "boites" différentes *)
```

[17]:
```
- : bool = false
```

L'égalité physique `==` est la même que celle de C ou Java, où elle s'écrit d'ailleurs de la même façon.

Attendons un peu pour comparer des références avec l'égalité "profonde" notée `=` en OCaml…

Pour lire la valeur contenue dans la case mémoire, on utilise l'opérateur `!` (prononcez "bang")

[18]:
```
!r  (* <=> r.(0) <=> r.content *)
```

[18]:
```
- : int = 42
```

[19]:
```
!r = !r2
```

[19]:
```
- : bool = true
```

Pour modifier la valeur contenue dans une référence, on utilise l'opérateur infixe `(:=)`.

[20]:
```
r := 43
```

[20]:
```
- : unit = ()
```

[21]:
```
r
```

[21]:
```
- : int ref = {contents = 43}
```

Notez que l'**affectation** `r:=e` est une instruction (une expression de type `unit`). L'affectation modifie l'état, mais "ne renvoie rien".

Sans surprise, on ne modifie que la valeur contenue à l'"adresse" `r`, mais pas aux autres adresses.

[22]:
```
r2
```

[22]:
```
- : int ref = {contents = 42}
```

[23]:
```
!r = !r2
```

```
[23]: - : bool = false
```

1.6 L'aliasing

Deux variables peuvent désigner une même référence.

```
[24]: let r' = r    (* r et r' sont deux alias pour une même référence *)
```

```
[24]: val r' : int ref = {contents = 43}
```

```
[25]: r == r' (* les variables r et r' désignent la même adresse mémoire *)
```

```
[25]: - : bool = true
```

```
[26]: r := 44 (* modifie la valeur à l'adresse égale à la valeur de r ... *)
```

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

```
[27]: !r' (* ... qui  était aussi l'adresse égale à la valeur de r' *)
```

```
[27]: - : int = 44
```

1.7 La gestion mémoire: soulevons (à peine) le capot...

Si on veut se représenter la situation par un dessin avec des boites, il faut comprendre qu'il y a deux niveaux de boites: - les boites des variables, qui contiennent la valeur des variables. La valeur d'une variable de type `'a ref` est une adresse - les boites à des adresses quelconques, qui contiennent la valeur désignée par la référence

Une valeur de la forme `ref 48` est donc une adresse mémoire d'une boite qui contient 48. OCaml ne permet pas d'afficher ou d'avoir une quelconque information sur cette adresse.

En fait rien ne dit même que cette adresse ne sera pas changée par OCaml en cours de programme!

Pourquoi OCaml changerait-il l'adresse d'une référence en mémoire?

C'est le **gestionnaire mémoire** d'OCaml (le *garbage collector*, GC) qui peut déplacer des références pendant l'exécution du programme, de manière transparente pour le programme.

Déplacer des références est utile pour lutter contre la **fragmentation**. Le chapitre 9 du livre Developing applications with Objective Caml est consacré au GC d'OCaml, n'hésitez pas à le lire pour approfondir le sujet...

1.8 Les séquences d'instructions

L'opérateur ; permet de composer une instruction avec une seconde expression. C'est cette seconde expression qui donne la valeur de l'expression globale

[28]:
```
let x = print_string "hello"; print_newline() ; 3
```

```
hello
```

[28]:
```
val x : int = 3
```

La valeur de x est 3, mais pendant son calcul, OCaml a affiché hello.

[29]:
```
let compteur = ref 0
```

[29]:
```
val compteur : int ref = {contents = 0}
```

[30]:
```
let suivant () = compteur := !compteur + 1 ; !compteur
```

[30]:
```
val suivant : unit -> int = <fun>
```

[31]:
```
let x1 = suivant () in let x2 = suivant () in let x3 = suivant () in
(x1, x2, x3)
```

[31]:
```
- : int * int * int = (1, 2, 3)
```

A chaque appel, `suivant` incrémente `compteur` par **effet de bord**.

La fonction `suivant` renvoie `!compteur`, qui est différent à chaque fois.

Pour faire la même chose en Python, on aurait utilisé une variable globale, et surtout on aurait eu besoin du mot-clé `return`.

```
compteur = 0
def suivant () :
    global compteur
    compteur += 1
    return compteur
```

La vie sans `return`

> Le mot-clé `return` n'existe pas en OCaml, simplement la valeur "renvoyée" par une suite d'expressions est la valeur de la dernière expression.

En fait, `e1; e2` est presque équivalent à `let () = e1 in e2`. Il y a une petite nuance toutefois…

[32]:
```
2 + 1 ; 3 + 2
(* évalue 2+1, jette le résultat, puis évalue 3+2 et "renvoie" le résultat *)
```

```
File "[32]", line 1, characters 0-5:
1 | 2 + 1 ; 3 + 2
    ^^^^^
Warning 10: this expression should have type unit.
File "[32]", line 1, characters 0-5:
1 | 2 + 1 ; 3 + 2
    ^^^^^
Warning 10: this expression should have type unit.
```

[32]:
```
- : int = 5
```

[33]:
```
let () = 2+1 in 3+2
```

```
File "[33]", line 1, characters 4-6:
1 | let () = 2+1 in 3+2
        ^^
Error: This pattern matches values of type unit
       but a pattern was expected which matches values of type int
```

Conclusion:
- dans `let () = e1 in e2`, `e1` doit être de type `unit` - dans `e1; e2`, si `e1` n'est pas de type `unit`, ça passe, mais on a droit à un avertissement.

Petit conseil, si OCaml vous affiche un warning à propos de `;`, c'est sûrement que votre programme a un petit soucis. Voici une erreur ultra classique.

[85]:
```
let x = print_string "hello"; print_newline; 3
```

[85]:
```
val x : int = 3
```

```
File "[85]", line 1, characters 30-43:
1 | let x = print_string "hello"; print_newline; 3
                                  ^^^^^^^^^^^^^
Warning 10: this expression should have type unit.
```

Voyez-vous pourquoi il y a un warning, et pourquoi hello ne s'est pas affiché?

1.9 Les blocs d'instructions

Le plus souvent il n'y a pas d'ambiguïté sur le début et la fin d'un bloc d'instructions. Cependant il est courant de marquer un bloc avec des parenthèses ou des `begin`/`end` pour plus de lisibilité.

Et dans certaines situations, marquer le bloc est vraiment nécessaire.

[90]:
```
let () = if 0=1 then print_string "hello "; print_string "world";␣
 ↪print_newline()
(* affiche world *)
```

```
world
```

[91]:
```
let () =
  if 1=1 then begin
    print_string "hello ";
    print_string "world";
    print_newline() (* <- le dernier ; est autorisé mais il n'est pas␣
 ↪nécessaire *)
  end
(* affiche hello world *)
```

```
hello world
```

[37]:
```
let () = if 0=1 then (
   print_string "hello";
   print_string "world";
   print_newline(); (* <- le dernier ; est autorisé mais il n'est pas␣
 ↪nécessaire *)
)
(* affiche hello world*)
```

1.10 Les boucles

Oui! Les boucles existent bien en OCaml. Elles ont une syntaxe légèrement inhabituelle… mais on s'y habitue!

[38]:
```
let tab5 = [|0;0;0;0;0|]  (* un tableau de 5 entiers *)
```

[38]:
```
val tab5 : int array = [|0; 0; 0; 0; 0|]
```

[39]:
```
for i=0 to 4 do   (* <- 4 inclus *)
  tab5.(i) <- i;
  print_int i; (* <- le dernier ; est autorisé mais il n'est pas nécessaire *)
done;
print_newline()
```

```
01234
```

[39]:
```
- : unit = ()
```

[40]:
```
for i=4 downto 0 do begin
  print_int i
end done;
print_newline()
```

```
43210
```

[40]:
```
- : unit = ()
```

[41]:
```
let i = ref 0
```

[41]:
```
val i : int ref = {contents = 0}
```

[42]:
```
while !i < 5 do print_int !i; i := !i + 1 done;
print_newline()
```

```
01234
```

[42]:
```
- : unit = ()
```

1.11 Style impératif et style récursif

Quand on doit écrire une fonction qui s'écrit naturellement avec une boucle dans un langage non fonctionnel, on peut l'écrire avec une boucle for ou while en OCaml.

Mais, comme on l'a déjà vu, on peut aussi l'écrire en utilisant une fonction récursive.

Comparons les deux styles sur un exemple: le calcul de la somme $f(0) + f(1) + \cdots + f(n-1)$ pour une fonction f donnée.

En style impératif, cela donne

[43]:
```
let somme f n =
  let res = ref 0 in
  for i=0 to n-1 do
    res := !res + f i
  done;
  !res (* return implicite, c'est la dernière valeur d'une séquence de ; *)
```

[43]:
```
val somme : (int -> int) -> int -> int = <fun>
```

En style récursif, on utilise une fonction récursive auxiliaire pour imiter la boucle for

[44]:
```
let somme f n =
  let rec somme_depuis i = (* f(i) + f(i+1) + ... + f(n-1) *)
    if i = n
```

```
    then 0
    else f i + somme_depuis (i+1)
  in somme_depuis 0
```

[44]:
```
val somme : (int -> int) -> int -> int = <fun>
```

1.12 La pile d'appel

Déroulons le calcul de `somme id 3`, où `id = fun x -> x`

L'appel récursif étant **enveloppé**, il faut mémoriser des calculs en attente. Ici, je mets en attente l'addition $i + \cdots$ à l'étape i.

Ces calculs en attente sont mémorisés dans la **pile d'appel**, dont nous avions déjà parlé en Python. Cette pile d'appel peut consommer rapidement beaucoup de mémoire, c'est pourquoi Python ne favorise pas la programmation par récurrence.

Et c'est aussi pour cela que, si l'on devait choisir entre la version impérative et la version récursive précédentes, il faudrait privilégier la version impérative.

Mais alors, les programmeurs Haskell et les puristes du style fonctionnel ont-ils tort de ne jamais programmer avec de boucles for et while?

1.13 Récursivité terminale (aka itération)

Il y a une autre façon d'écrire la fonction récursive `somme_depuis` utilisée pour calculer la fonction `somme`: on peut ajouter un accumulateur en paramètre.

[93]:
```
let somme f n =
  let rec somme_depuis i acc =
    if i = n
    then acc
    else somme_depuis (i+1) (acc + f i)
  in somme_depuis 0 0
```

[93]:
```
val somme : (int -> int) -> int -> int = <fun>
```

L'intérêt de cette définition est que la récurrence n'est plus enveloppée, on n'a plus besoin de mettre des calculs en attente.

On parle de **récursivité terminale**.

Observons la différence quand on développe le calcul de `somme id 3`

$$
\begin{aligned}
& \mathsf{somme}(\mathsf{id}, 3) \\
= \; & \mathsf{somme\_depuis}(0, 0) \\
= \; & \mathsf{somme\_depuis}(1, 0 + \mathsf{id}(0)) \\
= \; & \mathsf{somme\_depuis}(1, 0) \\
= \; & \mathsf{somme\_depuis}(2, 0 + \mathsf{id}(1)) \\
= \; & \mathsf{somme\_depuis}(2, 1) \\
= \; & \mathsf{somme\_depuis}(3, 1 + \mathsf{id}(2)) \\
= \; & \mathsf{somme\_depuis}(3, 3) \\
= \; & 3
\end{aligned}
$$

Les compilateurs des langages fonctionnels optimisent les fonctions récursives terminales: au lieu de générer du code pour un appel de fonction "classique", qui remplirait la pile d'appel, ils transforment le code en quelque chose de proche d'une boucle while.

C'est pourquoi quand vous lirez du code fonctionnel écrit par un adepte de programmation fonctionnelle, vous trouverez souvent des fonctions récursives là où vous vous attendriez à voir une boucle while.

pause musicale: https://www.youtube.com/watch?v=-PX0BV9hGZY

1.14 Appel par adresse

Certaines fonctions prennent en paramètre des mutables et "écrivent dedans". Un exemple classique est la fonction qui incrémente un compteur.

[46]:
```
let inc c = (* incrémente le compteur c*)
  c := !c + 1
```

[46]:
```
val inc : int ref -> unit = <fun>
```

[47]:
```
let c0 = ref 0
```

[47]:
```
val c0 : int ref = {contents = 0}
```

[48]:
```
inc c0
```

[48]:
```
- : unit = ()
```

[49]:
```
!c0
```

[49]:
```
- : int = 1
```

Et en Python?

L'appel par adresse ne concerne pas les variables. Les tableaux, en revanche, sont passés par adresse.

```
c = 0
def inc(c) : c = c + 1
inc(c)
print(c) # -> 0

c = [0]
def inc(c) : c[0] = c[0] + 1
inc(c)
print(c[0]) # -> 1
```

Autre exemple: l'échange de valeurs dans un tableau

[50]:
```
let swap tab i j =
   let tmp = tab.(i) in
   tab.(i) <- tab.(j);
   tab.(j) <- tmp
```

[50]:
```
val swap : 'a array -> int -> int -> unit = <fun>
```

[51]:
```
let tab3 = [|0; 1; 2|]
```

[51]:
```
val tab3 : int array = [|0; 1; 2|]
```

[52]:
```
swap tab3 1 2
```

[52]:
```
- : unit = ()
```

[53]:
```
tab3
```

[53]:
```
- : int array = [|0; 2; 1|]
```

1.15 Notion de clôture

Laissons un instant les références pour préciser un point sur les définitions de fonctions.

Le corps d'une fonction peut faire référence à des variables "globales": par exemple

[54]:
```
let x = 0                    (* une variable "globale" *)
let f = fun n -> x * n       (* une fonction qui utilise la variable globale x *)
```

[54]:
```
val x : int = 0
```

[54]:
```
val f : int -> int = <fun>
```

Que se passe-t-il si l'on redéfinit la variable globale `x`? La fonction `f` change-t-elle?

[55]:
```
let x = 1
(* redéfinition de la variable x; l'ancienne définition est "masquée" *)
```

[55]:
```
val x : int = 1
```

[56]:
```
f 1     (* f 1 = x * 1 ...  mais avec l'ancien x *)
```

[56]:
```
- : int = 0
```

On observe que le `x` utilisé par `f` n'est pas celui de la deuxième définition, mais celui de la première. Essayons de reconstituer ce qu'il s'est passé:

1. Lorsqu'on a défini `f`, OCaml a sauvegardé le contexte de définitions dans lequel il faut évaluer `x * n`.
2. Lorsque, plus tard, on a appellé la fonction `f`, on a restauré ce contexte de définitions.

C'est "comme si" on avait directement remplacé le `x` par sa valeur 0 au moment de la définition de `f`.

La variable `f` a pour valeur la fonction qui à `n` associe `x * n` dans le contexte de définitions où `x=0`. La valeur de `f` est appellée une **clôture** : une clôture est donc constituée de trois parties

- un identifiant de variable pour décrire le paramètre de la fonction (ici `n`)
- une expression (ici `x * n`)
- un contexte de définitions (ici restreint à une seule définition, `x=0`)

1.16 Clôtures et références

Revenons-en maintenant aux références, et leur utilisation combinée aux clôtures.

[ ]:
```
let next =
  let n = ref 0 in
  fun () -> n:= !n + 1 ; !n
```

[58]:
```
for i=1 to 5 do print_int (next ()); print_newline() done
```

```
1
2
3
4
5
```

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

Notre fonction `next` ressemble beaucoup à la fonction `suivant` vue tout à l'heure:

```
let compteur = ref 0
let suivant () = compteur:= !compteur + 1; !compteur

let next =
  let n = ref 0 in
  fun () -> n:= !n + 1; !n
```

La différence entre `suivant` et `next` est la suivante: `suivant` utilise une variable "globale" (avec des guillements) `compteur` que tout le monde peut modifier, tandis que `next` utilise une variable `n` "locale" et surtout "privée": il est impossible de lire ou modifier le contenu de la référence `n` autrement qu'en appelant `next`.

Le mot-clé `static` permet de faire quelque chose d'un peu similaire en C (une variable "à portée locale" mais à durée de vie "globale", qui "survit" aux appels de fonction).

```
int suivant() {
  static int n = 0;  /* variable locale à durée de vie globale, initialisée en début de progra
  n++;
  return n;
}
```

1.17 `fun`, `function`, ou `let f x = ...` ?

Nous avons vu plusieurs façons de définir des fonctions: récapitulons

```
[59]: let f1 x = x + 1
      let f2 = function x -> x + 1
      let f3 = fun x -> x + 1
```

```
[59]: val f1 : int -> int = <fun>
```

```
[59]: val f2 : int -> int = <fun>
```

```
[59]: val f3 : int -> int = <fun>
```

En fait `let f x1 .. xn = e` est complètement équivalent à `let f = fun x1 .. xn -> e`.

Par ailleurs, `function ...` est un racourci pour `fun x -> match x with ...`. On utilise donc le plus souvent `function` à la place d'un `match/with`. En revanche, pour définir une fonction anonyme, on utilise plutôt `fun`. D'ailleurs, on ne peut pas écrire une fonction anonyme à plusieurs variables avec `function`.

[60]:
```
fun x y -> x+y
```

[60]:
```
- : int -> int -> int = <fun>
```

[61]:
```
function x y -> x+y
```

```
File "[61]", line 1, characters 11-12:
1 | function x y -> x+y
                 ^
Error: Syntax error
```

1.18 Références et polymorphisme

Avant de passer au dernier sujet de ce cours, évoquons un point un peu subtil sur le typage des références. Il s'agit plus de collecter quelques observations que d'expliquer les raisons profondes de cette subtilité.

On a déjà rencontré plusieurs valeurs *polymorphes* dont le type est paramétré par une ou plusieurs variables de type, par exemple

- `(fun x-> x):'a->'a`
- `[]:'a list`.

Ces valeurs peuvent être utilisées dans différents contexte en instanciant de plusieurs façons la variable de type.

[62]:
```
let f = fun x -> x
let nil = []
let _ = (f 1, f "toto", 1::nil, "toto"::nil)
```

[62]:
```
val f : 'a -> 'a = <fun>
```

[62]:
```
val nil : 'a list = []
```

[62]:
```
- : int * string * int list * string list = (1, "toto", [1], ["toto"])
```

Si l'on manipule ces valeurs au travers d'une référence, la variable de type se présente un peu différemment: ce n'est plus `'a` mais `'_weak`

[63]:
```
let f = ref (fun x -> x)
let nil = ref []
```

[63]:
```
val f : ('_weak1 -> '_weak1) ref = {contents = <fun>}
```

[63]:
```
val nil : '_weak2 list ref = {contents = []}
```

On parle ici de type **monomorphe**: les variables `_weak` ne peuvent pas être remplacées tantôt par `int` ou `string` comme ce serait le cas avec un type polymorphe.

La variable de type `_weak` désigne un type que l'inférence de type n'a pas encore réussi à déterminer.

C'est la première utilisation de la référence qui permettra de déterminer ce type, et on ne pourra plus le changer ensuite.

[64]:
```
!f 1  (* j'utilise !f avec un entier: donc _weak1=int *)
```

[64]:
```
- : int = 1
```

[65]:
```
let toto () = !f "toto"   (* trop tard, !f est maintenant de type int -> int *)
```

```
File "[65]", line 1, characters 17-23:
1 | let toto () = !f "toto"   (* trop tard, !f est maintenant de type int -> in
 ↪*)
                     ^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

Il faut avoir en tête que la fonction `toto` peut être appelée plus tard, en particulier après avoir effectué l'affectation `f:= fun x -> x+1`… Ce serait problématique de passer `"toto"` comme argument à `fun x -> x+1`.

Cet autre exemple illustre le problème d'une autre façon.

[66]:
```
let liste = 1 :: !nil   (* j'utilise nil comme une référence sur une liste
 ↪d'entiers *)
```

[66]:
```
val liste : int list = [1]
```

[67]:
```
nil := ['#']
(* je n'ai plus le droit de l'utiliser comme une référence
   sur une liste de caractères *)
```

```
File "[67]", line 1, characters 8-11:
1 | nil := ['#']  (* je n'ai plus le droit de l'utiliser comme une référence su
 ↪une liste de caractères *)
            ^^^
```

```
Error: This expression has type char but an expression was expected of type
         int
```

Plutôt normal, on n'a pas envie que `liste` devienne `[1; '#']`, ce serait une liste mal typée…

1.19 Résumons

On peut programmer de manière impérative en OCaml:

- on simule une variable "mutable" d'un programme impératif avec une référence
- on peut utiliser des boucles for et while
- on n'a pas besoin de return, il suffit de placer la valeur que l'on veut renvoyer à la fin de la suite d'instructions
- si on veut simuler un return pour interrompre une boucle, il faudra utiliser une exception (prochain cours)