Chapitre 4
Programmation impérative

1. Introduction

1.1 Programmation par effets

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. Si on se réfère au modèle d'architecture de von Neumann étudié en cours d'informatique de tronc commun (voir figure 1), on observe que le processeur peut agir sur

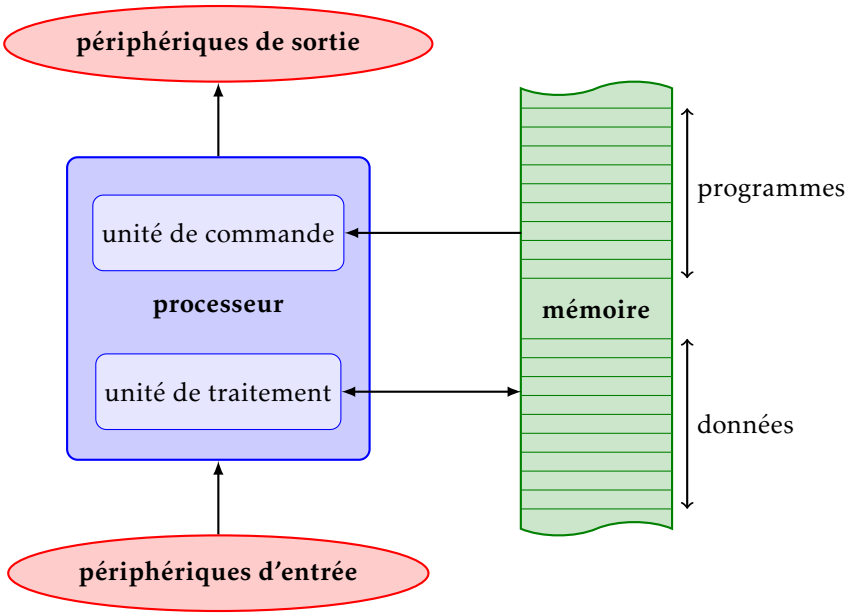


Figure 1 – L'architecture d'un ordinateur selon le modèle de von Neuman.

l'état des périphériques d'entrée/sortie et sur l'état de la mémoire. On distinguera donc deux types d'*effets* :

- les *effets de bords* lorsque le processeur réalise une intervention avec l'extérieur (entrée ou sortie) ;
- les *effets* à proprement parler, qui modifient l'état de la mémoire.

• Séquence d'instructions

En Caml, les séquences d'instructions sont des expressions séparées par des points-virgules. L'évaluation de la séquence :

```
expr1 ; expr2 ; ... ; exprn
```

provoque les effets éventuels de ces n expressions dans l'ordre indiqué et retourne le résultat de la dernière, le résultat des autres expressions étant supprimé.
De fait, utiliser une séquence d'instructions sans effets ne présente aucun intérêt, ce qui explique pourquoi nous n'en avons pas eu l'usage jusqu'à présent. On observera aussi que la plupart du temps, ces différentes expressions sont (à l'exception peut-être de la dernière) de type *unit* puisque toute expression doit être typée mais que le résultat est ignoré.

• Effets de bords

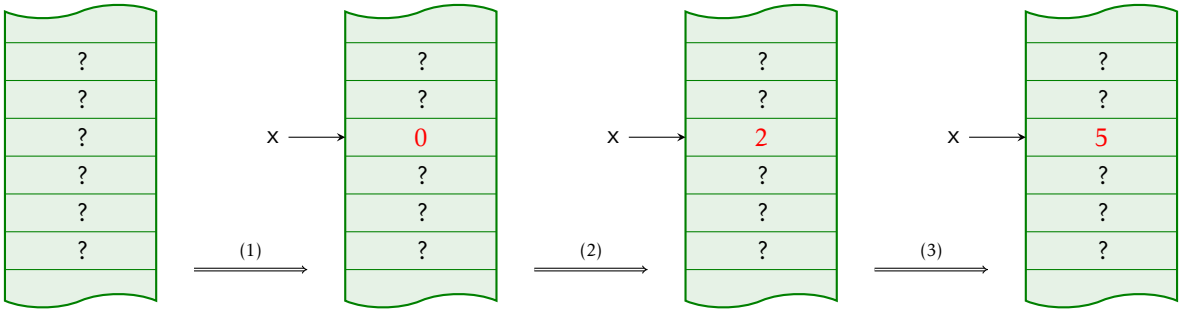
Nous avons déjà brièvement rencontré des instructions réalisant des effets de bords : les fonctions **`print_int`**, **`print_float`**, **`print_char`**, **`print_string`** et **`print_newline`** agissent (en sortie) sur l'écran. On peut noter qu'elles ont toutes un type de la forme *type –> unit*.
Nous aurons l'occasion plus tard d'utiliser d'autres instructions permettant d'interagir avec une fenêtre graphique.

De manière symétrique, les fonctions **`read_int`**, **`read_float`** et **`read_line`** agissent en entrée en demandant à l'utilisateur d'utiliser son clavier pour envoyer au processeur respectivement un entier, un flottant ou une chaîne de caractères. Ces trois instructions ont un type de la forme *unit –> type*.
Il existe naturellement aussi des instructions permettant d'interagir avec la souris.

1.2 Références

Nous l'avons dit, en programmation impérative le programmeur agit directement sur la mémoire de la machine. Pour éviter une manipulation fastidieuse des adresses, on associe à un emplacement de la mémoire un *identifiant* qui la caractérise : c'est la création d'une *référence*.
En Caml, on crée une référence à l'aide de la construction : **`let identifiant = ref valeur`** en spécifiant l'identifiant de cette référence et la valeur qu'elle contient initialement. Il faudra bien distinguer le contenant (la référence) de son contenu (la valeur), qu'on obtient à l'aide de l'opérateur de déréférencement (le point d'exclamation) : si **`x`** désigne une référence, **`!x`** désigne la valeur contenue dans celle-ci.
Une fois créé, le contenu d'une référence peut être modifié à l'aide de l'opérateur infixe d'affectation **`:=`**

```
# let x = ref 0 ;;                    (1)
x : int ref = ref 0
# x := 2 ;;                           (2)
- : unit = ()
# x := !x + 3 ;;                      (3)
- : unit = ()
# !x ;;                               (4)
- : int = 5
# x ;;                                (5)
- : int ref = ref 5
```



Comme on peut le noter sur cet exemple ligne 5, le type d'une référence est *'a ref*, où *'a* désigne le type de son contenu.

• Différence entre une définition et une référence

Nous l'avons déjà dit, 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 l'exemple ci-dessous :

```
# let a = 1 ;;
a : int = 1
# let incr x = x + a ;;
incr : int -> int = <fun>
# let a = 2 ;;
a : int = 2
# incr 0 ;;
- : int = 1
```

```
# let a = ref 1 ;;
a : int ref = ref 1
# let incr x = x + !a ;;
incr : int -> int = <fun>
# a := 2 ;;
- : unit = ()
# incr 0 ;;
- : int = 2
```

- **Égalité physique et structurelle**

De cette différence entre contenant et contenu résulte l'existence de deux sortes d'égalité en Caml : l'égalité *structurelle* qui teste l'égalité entre deux valeurs, qui est représentée par le symbole `=`, et l'égalité *physique* qui détermine si les objets occupent les mêmes emplacements en mémoire, qui est représentée par le symbole `==`.

```
# let x = ref 0 ;;
x : int ref = ref 0
# let y = ref 0 ;;
y : int ref = ref 0
# !x = !y ;;              (* s'évalue en 0 = 0 *)
- : bool = true
# !x == !y ;;
- : bool = true
# x = y ;;                (* s'évalue en ref 0 = ref 0 *)
- : bool = true
# x == y ;;
- : bool = false
```

`!x` et `!y` sont structurellement et physiquement égaux à la valeur 0.
`x` et `y` sont structurellement égaux car ils admettent tous deux pour valeur `ref 0`, mais physiquement différents car se trouvant à des emplacements en mémoire distincts.

L'égalité physique permet d'illustrer de nouveau le comportement dynamique d'une référence :

```
# let z = x ;;
z : int ref = ref 0
# z == x ;;               (* z est physiquement égal à x *)
- : bool = true
# x := 1 ;;
- : unit = ()
# !z ;;
- : int = 1
```

1.3 Types mutables

Lors de la définition d'un type enregistrement, certains champs, à l'instar des références, peuvent être déclarés *mutables* : le vérificateur de type autorise alors la modification physique du contenu du champ des objets de ce type.
Dès lors, il apparaît qu'une référence n'est qu'un enregistrement à un champ mutable ; on peut donc très facilement les redéfinir :

```
# type 'a myref = {mutable Valeur: 'a} ;;
Le type myref est défini.
```

Si `x` est un objet de type *'a ref*, on accède à la valeur qui lui est associée par la commande `x.Valeur` (l'équivalent de l'opérateur préfixe `!`) :

```
# function x -> x.Valeur ;;
- : 'a myref -> 'a = <fun>
```

Modifier le contenu du champ mutable se réalise à l'aide de l'opérateur infixe `<-` (l'équivalent de l'opérateur infixe `:=`) :

```
# fun x a -> x.Valeur <- a ;;
- : 'a myref -> 'a -> unit = <fun>
```

Illustrons enfin l'utilisation de ce nouveau type à l'aide des quelques lignes qui suivent :

```
# let x = {Valeur = 0} ;;        (* l'équivalent de : let x = ref 0 *)
x : int myref = {Valeur = 0}
# x.Valeur <- x.Valeur + 1 ;;    (* l'équivalent de : x := !x + 1 *)
- : unit = ()
# x.Valeur ;;                    (* l'équivalent de : !x *)
- : int = 1
```

2. Instructions à caractère impératif

2.1 Instruction conditionnelle

Une instruction conditionnelle, c'est à dire dont l'exécution dépend d'une condition, est construite en suivant le schéma :

`if condition then expr1 else expr2`

et signifie que si la condition est vérifiée, l'expression 1 sera évaluée, alors que dans le cas contraire c'est l'expression 2 qui le sera.
Remarquons qu'à l'instar de toute phrase en Caml, cette construction calcule une valeur ; en conséquence de quoi les deux expressions qui interviennent dans cette instruction *doivent être du même type* pour respecter la règle de typage strict de Caml :

```
# let x = if true then 1 else 2 ;;
x : int = 1
# if true then 1 else 2. ;;
Entrée interactive:
>if true then 1 else 2. ;;
>                   ^^
Cette expression est de type float,
mais est utilisée avec le type int.
```

Pour les mêmes raisons, on ne peut avoir de **`if ... then ...`** sans **`else`** (en effet, quel serait le type de l'expression dans le cas où la condition n'est pas vérifiée ?). Seule exception, lorsque la première expression est de type *unit* (dans ce cas, Caml ajoute implicitement **`else ()`** si vous ne le précisez pas vous-même).

```
# if true then 1 ;;
Entrée interactive:
>if true then 1 ;;
>             ^
Cette expression est de type int,
mais est utilisée avec le type unit.
# if true then print_int 1 ;;
1- : unit = ()
```

Notons que si on souhaite remplacer **`expr1`** et **`expr2`** par des séquences d'instructions, il est nécessaire d'encadrer celles-ci par les mots-clés **`begin`** et **`end`** ou à les encadrer par des parenthèses.

2.2 Instructions d'itération

Caml dispose de deux sortes de boucles pour répéter des effets : les boucles inconditionnelles, qui répètent un calcul un nombre de fois fixé à l'avance, et les boucles conditionnelles, qui répètent un calcul tant qu'une condition est vérifiée.

• Boucles inconditionnelles

Il y a deux types de boucles inconditionnelles, qui suivent l'un des deux schémas suivant :

```
for ident = expr1 to expr2 do expr3 done
for ident = expr2 downto expr1 do expr3 done
```

Dans le premier cas, l'indice de boucle (forcément de type entier) est incrémenté de 1 en 1 entre les valeurs prises par expr1 et expr2 en effectuant à chaque fois le calcul de expr3 sous forme de séquence. Bien entendu, si expr3 n'a pas d'effet, cette instruction n'a pas grand sens.
Dans le second cas, c'est la même chose, sauf que l'indice de boucle est cette fois décrémenté de 1 en 1.
Par exemple :

```
# for i = 10 downto 1 do print_int i ; print_char ' ' done ;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

• Boucle conditionnelle

La boucle conditionnelle suit le schéma suivant :

```
while expr1 do expr2 done
```

et signifie qu'on évalue expr2 tant que la condition expr1 est vérifiée (ce qui impose à cette expression d'être de type *bool*). Là encore, si expr2 ne réalise pas d'effet, on voit mal comment la condition pourrait cesser d'être vérifiée...

```
# let x = ref 10 in
   while !x > 0 do print_int !x ; print_char ' ' ; x := !x - 1 done ;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

3. Tableaux

Les tableaux, appelés aussi vecteurs en Caml, sont des suites finies et modifiables de valeurs d'un même type. Si *type* désigne le type de ces valeur, celui d'un tableau sera *type vect*.

3.1 Construction et manipulation d'un tableau

Il y a deux manières de définir un tableau :

- en dressant la liste de ses éléments, séparés par un point-virgule et encadrés par les symboles `[|` et `|]` ;
- ou en créant le tableau à l'aide de la fonction **`make_vect`** et en le remplissant ultérieurement.

La fonction **`make_vect`** a pour objet de créer un tableau en donnant sa taille et un élément qui sera mis au départ dans toutes les cases de ce tableau (sa valeur d'initialisation). C'est une fonction de type *int -> 'a -> 'a vect*.
Par exemple :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] and u = make_vect 5 1 ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
u : int vect = [|1; 1; 1; 1; 1|]
```

Une fois créé, on peut consulter et modifier le contenu de ses cases : si **`t`** est un tableau et **`k`** un entier, **`t.(k)`** désigne le contenu de la case d'indice k. Attention,

dans un tableau de taille n, les indices s'échelonnent entre 0 et $n-1$.

On affecte la valeur **`v`** à la case d'indice **`k`** d'un tableau par la construction **`t.(k) <- v`**. Par exemple :

```
# u.(1) <- 2 ;;
- : unit = ()
# u.(2) <- u.(1) + 1 ;;
- : unit = ()
# u ;;
- : int vect = [|1; 2; 3; 1; 1|]
```

Notons enfin que la fonction **vect_length**, de type *'a vect –> int*, retourne la taille d'un tableau :

```
# vect_length u ;;
– : int = 5
```

- **Parcours d'un tableau**

On observera sans surprise que tableaux et boucles font bon ménage. Par exemple, pour calculer le plus grand élément d'un tableau, on utilisera une boucle inconditionnelle, puisqu'il est nécessaire de parcourir le tableau dans son entier :

```
let max_vect t =
  let n = vect_length t in
    if n = 0 then raise Not_found
    else let m = ref t.(0) in
      for k = 1 to n-1 do m := max !m t.(k) done ;
  !m ;;
```

En revanche, pour déterminer s'il existe un élément d'un tableau vérifiant une certaine propriété, on utilise une boucle conditionnelle (il n'est pas nécessaire de parcourir le tableau dans son entier dès lors qu'on a trouvé une solution) :

```
let cherche prop t =
  let rep = ref false and k = ref 0 in
    while !k < vect_length t && not !rep do rep := prop t.(!k) ;
                                            k := !k + 1  done ;
  !rep ;;
```

Mais notons bien que rien ne nous oblige à adopter un style de programmation impératif lorsqu'on manipule un tableau. On pourra d'ailleurs préférer la version fonctionnelle de ces deux fonctions :

```
let max_vect t =
    let rec aux = function
      | i when i < 0 -> raise Not_found
      | i when i = 0 -> t.(0)
      | i            -> max t.(i) (aux (i-1))
    in aux (vect_length t - 1) ;;
```

La fonction récursive auxiliaire calcule le maximum de $[|t_0;\cdots;t_i|]$.

```
let cherche prop t =
    let rec aux = function
      | i when i < 0 -> false
      | i            -> prop t.(i) || aux (i-1)
    in aux (vect_length t - 1) ;;
```

Ici, la fonction auxiliaire détermine si l'un des éléments du tableau $[|t_0;\cdots;t_i|]$ vérifie la propriété, en utilisant le principe de l'évaluation paresseuse.

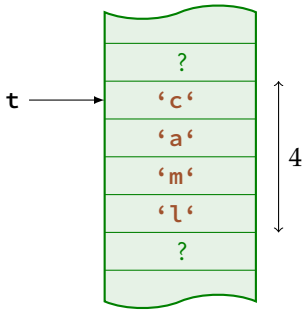
3.2 Comparaison entre listes et tableaux

Les tableaux, comme les listes, sont des structures de données linéaires qui modélisent les suites finies et homogènes de valeurs. Elles n'en sont pas moins très différentes. En effet, nous avons vu qu'insérer un élément en tête de liste a un coût constant mais qu'à l'inverse, accéder à un élément qui ne se trouve pas en tête de liste ne peut se faire sans parcourir tous les éléments qui le précèdent.
Pour les tableaux, c'est le contraire : on accède à un élément d'un tableau avec le même coût quel que soit l'emplacement de cet élément, mais à l'inverse, augmenter ou diminuer la taille d'un tableau a un coût proportionnel à la taille de celui-ci.

Pour comprendre pourquoi, il est nécessaire de parler de nouveau de machine abstraite. Lorsqu'on crée un tableau, nous avons vu qu'il est nécessaire de spécifier sa taille n et son contenu initial. Dans la mémoire, un

bloc de n emplacements *consécutifs* est alors associé à ce tableau [1] ; ainsi, connaissant l'adresse du premier emplacement et la taille de ceux-ci, un simple calcul arithmétique donne l'adresse de l'élément auquel on souhaite accéder :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
```



En contrepartie, les emplacements mémoire qui précèdent et succèdent ceux occupés par le tableau peuvent être utilisés à d'autres usages. C'est pourquoi il est nécessaire, lorsqu'on souhaite modifier la taille d'un tableau, de créer un nouveau tableau, de taille plus grande ou plus petite, puis de copier tous les éléments du tableau original dans le nouveau tableau (et éventuellement libérer l'espace mémoire alloué à l'ancien tableau).

3.3 Tableaux bi-dimensionnels

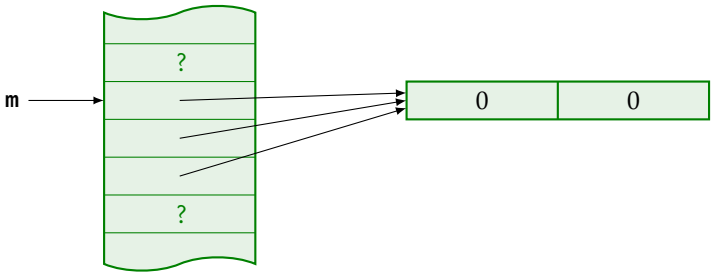
Essayons maintenant de créer un tableau bi-dimensionnel, par exemple ayant trois lignes et deux colonnes, en procédant de la façons suivante :

```
# let m = make_vect 3 (make_vect 2 0) ;;
m : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

et modifions la première case de ce tableau :

```
# m.(0).(0) <- 1 ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|1; 0|]; [|1; 0|]; [|1; 0|]|]
```

Cela ne répond pas à notre attente et pourtant, on pouvait s'y attendre : c'est la même valeur physique, à savoir `make_vect 2 0` qui a été attribuée à chacune des trois cases de notre tableau `m`. On peut donc représenter la construction que nous avons faite de la manière suivante :

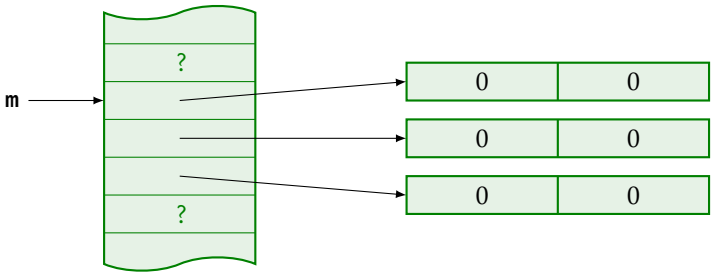


Nous pouvons apporter une première solution à ce problème en procédant en deux temps :

```
# let m = make_vect 3 [||] ;;
 m : '_a vect vect = [|[||]; [||]; [||]|]
# for i=0 to 2 do m.(i) <- make_vect 2 0 done ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

1. en réalité $n+1$, car dans le premier emplacement est stockée la valeur de la longueur du tableau.

Nous avons cette fois réalisé la construction suivante, qui répond à notre attente :



Il existe heureusement une instruction qui permet de réaliser cette construction plus rapidement, la fonction **make_matrix**, de type *int –> int –> 'a –> 'a vect vect* qui renvoie un tableau bi-dimensionnel. Par exemple, pour construire un tableau à trois lignes et deux colonnes et rempli initialement par la valeur 0, il suffit d'écrire :

```
# let m = make_matrix 3 2 0 ;;
m : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

L'élément de la $(i+1)^e$ ligne et de la $(j+1)^e$ colonne se trouve alors à l'emplacement **m.(i).(j)**, et c'est une donnée mutable.

3.4 Chaînes de caractères

Terminons avec quelques mots au sujet des chaînes de caractères, qui sont elles aussi des données mutables [2], comme le prouvent les lignes suivantes :

```
# let s1 = "abc" and s2 = "abc" in s1 == s2 ;;
- : bool = false
```

On ne s'étonnera donc pas qu'une chaîne de caractères ne soit guère différente d'un tableau de caractères de type *char vect*. Seuls vont différer les noms des instructions : **string_length** au lieu de **vect_length**, **make_string** au lieu de **make_vect**, et **s.[i]** pour désigner le $(i+1)^e$ caractère d'une chaîne, au lieu de **s.(i)**.

```
# let s = "azerty" in
    s.[0] <- 'q' ;
    s.[1] <- 'w' ;
    s ;;
- : string = "qwerty"
```

4. Exercices

4.1 Programmation impérative

Exercice 1

a) Définir une fonction **swap** de type *'a ref –> 'a ref –> unit* qui permute le contenu de deux références de même type.

b) Déterminer le type et le rôle de la fonction suivante :

```
let f x y =
  x := !x + !y ;
  y := !x - !y ;
  x := !x - !y ;;
```

Qu'en pensez-vous ?

2. À noter, les auteurs du langage ont ensuite modifié ce choix : en OCaml les chaînes de caractères ne sont plus des données mutables, ce qui correspond au choix le plus fréquent des langages de programmation.

Exercice 2 Calculer le n^e terme de la suite de Fibonacci dans un style impératif.

Exercice 3 Écrire dans un style impératif une fonction qui calcule l'image miroir d'une liste.

Exercice 4 Il n'existe pas de fonction Caml calculant x^n lorsque x est de type *int*, on se propose d'en définir une.

a) En utilisant la relation $x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{sinon} \end{cases}$, rédiger une fonction effectuant ce calcul, dans un style fonctionnel puis dans un style impératif. Combien de multiplications sont effectuées pour calculer x^n ?

b) Reprendre ces questions en utilisant cette fois les relations :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x \cdot x)^p & \text{si } n = 2p \\ x \cdot (x \cdot x)^p & \text{si } n = 2p+1 \end{cases}$$

Exercice 5 Déterminer le type et le rôle de la fonction suivante, puis la réécrire dans un style fonctionnel :

```
let f a b =
  let u = ref a and v = ref b in
    while !v <> 0 do
      let w = !v in v := !u mod !v ;
                    u := w
    done ;
    !u ;;
```

4.2 Tableaux et chaînes de caractères

Exercice 6 Définir les fonctions `map_vect` et `do_vect` qui agissent sur les tableaux à l'image des fonctions `map` et `do_list` sur les listes.

Exercice 7 Un *palindrome* est une chaîne de caractères qui peut être lue indifféremment de gauche à droite ou de droite à gauche en conservant le même sens. À titre d'exemple, on peut citer :

- `eluparcettecrapule` (élu par cette crapule) ;
- `tulastropecrasecesarceportsalut` (tu l'as trop écrasé, César, ce Port Salut).

L'objectif de cet exercice est d'écrire deux versions d'une fonction `palindrome`, de type *string –> bool*, testant si une chaîne de caractère est un palindrome.
La première version devra être écrite dans un style impératif :

> un mot $a_0 \cdots a_{n-1}$ est un palindrome si et seulement si $a_0 = a_{n-1}$, $a_1 = a_{n-2}$, etc.

La seconde version sera écrite dans un style fonctionnel :

> un mot $a_0 \cdots a_{n-1}$ de longueur supérieure ou égale à 2 est un palindrome si et seulement si $a_0 = a_{n-1}$ et si $a_1 \cdots a_{n-2}$ est un palindrome.

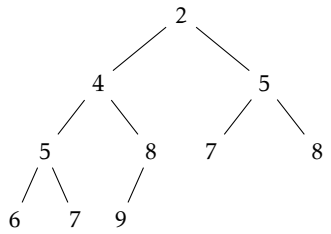
Exercice 8 Dans cet exercice, on représente un polynôme à coefficients entiers $\mathrm{P} = \sum_{k=0}^{n-1} a_k \mathrm{X}^k$ par le tableau de ses coefficients $[|a_0; a_1; \cdots; a_{n-1}|]$, et on souhaite, étant donné $b \in \mathbb{Z}$, calculer $\mathrm{P}(b)$.

a) La méthode naïve consiste à itérer la suite $u_1, u_1, \ldots, u_n$ définie par $u_p = \sum_{k=0}^{p-1} a_k b^k$. Rédiger une fonction qui calcule $\mathrm{P}(b)$ par cette méthode. Combien d'additions et de multiplications effectue-t-elle ?

b) La méthode de Hörner est basée sur l'égalité suivante : $\mathrm{P}(b) = a_0 + b\left(\sum_{k=0}^{n-2} a_{k+1} b^k\right)$. Rédiger une fonction qui calcule $\mathrm{P}(b)$ par cette méthode. Combien d'additions et de multiplications effectue-t-elle ?

c) Écrire enfin une fonction qui calcule le polynôme $\mathrm{P}(\mathrm{X} + b)$.

Exercice 9 La numérotation Sosa-Stradonitz des nœuds et feuilles d'un arbre binaire strict, dont nous avons parlé dans le module précédent, permet de représenter un arbre par un tableau, l'indice d'un élément correspondant à son numéro (par convention, l'élément d'indice 0 du tableau ne sera donc pas utilisé). Par exemple, l'arbre dont la représentation suit sera représenté par le vecteur [|?; 2; 4; 5; 5; 8; 7; 8; 6; 7; 9|] :

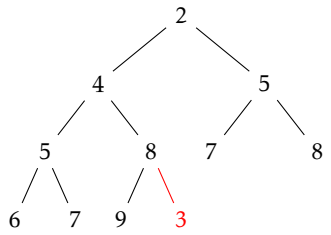


Évidemment, pour certains arbres, une telle représentation peut comporter de très nombreuses cases vides, aussi on la réserve pour des arbres *parfaits*, c'est à dire des arbres dont tous les niveaux sont remplis sauf éventuellement le dernier, partiellement rempli de la gauche vers la droite (ce qui est le cas de l'arbre ci-dessus). Avec cette exigence, le tableau associé ne comporte aucune case vide.

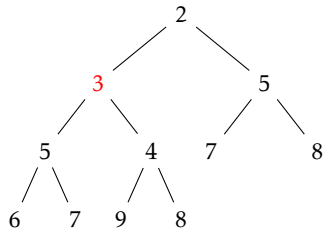
Un arbre parfait est appelé un *tas* (*heap* en anglais) lorsque les étiquettes des nœuds et feuilles appartiennent à un ensemble ordonné de sorte que l'étiquette du père soit toujours inférieure ou égale à celles de ses fils (ce qui est le cas de l'arbre donné en exemple).

a) Écrire une fonction `est_un_tas` qui vérifie si un tableau donné est la représentation d'un tas.

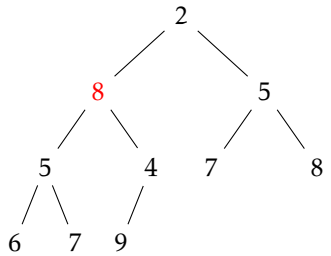
b) Pour insérer un nouvel élément dans un tas, on le place dans le premier emplacement disponible :



En général, l'arbre obtenu n'est plus un tas. Écrire une fonction qui « répare » un tel arbre pour en faire de nouveau un tas. Par exemple, l'arbre précédent sera corrigé pour donner le tas suivant :



c) À l'inverse, lorsqu'on supprime un élément dans un tas, on le remplace par le dernier élément du tableau. Par exemple, si on supprime l'élément que nous venons d'ajouter à notre tas, on obtient l'arbre suivant :



De nouveau, un tel arbre n'est en général plus un tas ; écrire une fonction (prenant en paramètre le numéro de l'élément déplacé) qui répare un tel arbre pour en faire de nouveau un tas.