Types et Structures de données
Notes de cours

P. Manoury

16 septembre 2003

Ce cours présente les structures de données devenues classiques en programmation : enregistrements, listes, tables, arbres, graphes.
On y étudiera leur mise en oeuvre dans un langage fortement typé aussi bien du point de vue fonctionnel qu'impératif. On introduira en particulier l'utilisation des modules pour la spécification et l'implantation de types de données abstraits.

Le langage support de ce cours est Objective Caml (`http ://caml.inria.fr`)

Plan du cours :

1. Éléments de langage et types de base (2 semaines).
2. Entrées/sorties (1 semaine).
3. Structures de données (1 semaine).
4. Types abstraits, modules (1 semaine).
5. Types algébriques, types somme (1 semaines).
6. Structures de données standard (1 semaine).
7. Structures chaînées (1 semaine).
8. Structures d'arbres (2 semaines).
9. Graphes (2 semaines).

Sources bibliographiques

- A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1985
- E. Chailloux, P. Manoury, B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly-France, 2000.
- C. Froidevaux, M-C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill Paris, 1990.
- X. Leroy *et al.*, *The Objective Caml system, release 3.06. Documentation and user's manual*, INRIA, 2002, `http ://caml.inria.fr`

1 Éléments du langage et types de base

1.1 Types et opérateurs numériques

- Entiers :
  - type `int`
  - valeurs $[-2^{30}, 2^{30}-1]$ (sur machine 32 bits)
  - notation `1 -1 0x10 0xffffffff 0o10 0b10`
- Flottants :
  - type `float`
  - valeurs mantisse 53 bits, exposant $[-1022, 1023]$ (norme IEEE 754)
  - notation `1.0 1. 0.1 1.e-1`

Entiers		Flottants	
`+`	addition	`+.`	addition
`-`	soustraction	`-.`	soustraction
`*`	multiplication	`*.`	multiplication
`/`	division entière	`/.`	division
`mod`	modulo	`**`	exponentiation

Expressions infixes, opérateurs à précédance :

```
          (1 + 2 * 3 mod 4)   =   (1 + ((2 * 3) mod 4))
 (1.0 +. 2.0 *. 3.0 ** 4.0)   =   (1.0 +. (2.0 *. (3.0 ** 4.0)))
```

Utilisation de la *boucle d'interaction* (*toplevel*) :

```
bash-2.05b: ocaml
        Objective Caml version 3.06

# (1 + 2 * 3 mod 4) ;;
- : int = 3
# (1.0 +. 2.0 *. 3.0 ** 4.0) ;;
- : float = 163.
#
```

1. On tape l'expression aprés le `#` (prompt)
2. On finit la *phrase* par `;;`
3. On obtient en retour (`- :`)
   - le *type* (`int` ou `float`)
   - la *valeur* (`3` ou `163.`)

Division par 0 : erreur (*exception*) ou valeur spéciale

```
# 1 / 0 ;;
Uncaught exception: Division_by_zero.
# 1.0 /. 0.0 ;;
- : float = Inf
```

Typage expressions *typées statiquement* (i.e. *avant* d'être évaluées)

```
# 1.0 / 0 ;;
This expression has type float but is here used with type int
```

```
# 1.0 /. 0 ;;
This expression has type int but is here used with type float
```

Fonctions de conversion explicite (lire la doc., remarquer le type) :

`val float_of_int : int -> float`
Convert an integer to floating-point.

`val int_of_float : float -> int`
Truncate the given floating-point number to an integer. The result is unspecified if it falls outside the range of representable integers.

Application *préfixe* :

```
# (float_of_int 1) ;;
- : float = 1.
# (float_of_int 1) /. 2.0 ;;
- : float = 0.500000
```

Limites calculs entiers *modulo*

```
# max_int ;;
- : int = 1073741823
# min_int ;;
- : int = -1073741824
# max_int+1;;
- : int = -1073741824
```

Attention aux débordements (non spécifié) :

```
# (float_of_int max_int) ** 2. ;;
- : float = 1152921502459363328.000000
# int_of_float ;;
- : float -> int = <fun>
# int_of_float ((float_of_int max_int) ** 2.) ;;
- : int = 0
```

1.2 Définir

Nommer une valeur :

```
# let pi = 3.1416 ;;
val pi : float = 3.1416
```

Définir une fonction :

```
# let aire_cyclindre r h =
    pi *. r**2. *. h
;;
val aire_cyclindre : float -> float -> float = <fun>
```

Remarquez : les fonctions sont traitées *comme des valeurs*.

Remarquez : pas de *return*, on écrit simplement l'expression qui donne le résultat de la fonction.

Application préfixe :

```
# (aire_cyclindre 1.2 7.) ;;
- : float = 31.667328
```

Syntaxe: `let` *name* $[arg_1 \ \ldots \ arg_k]$ = *expr*

Environnement, valeurs une déclaration ajoute un couple (nom, valeur) dans l'*environnement* du programme. La *valeur d'une fonction* est appelée *fermeture*, c'est un couple (code, environnement).

Définir localement

Syntaxe: `let` *name* [*arg*$_1$ ... *arg*$_k$] `=` *expr*$_1$ `in` *expr*$_2$

Exemple :

```
let aire_cyclindre r h =
  let pi = 3.1416 in
    pi *. r**2. *. h
;;
```

Remarque : une construction `let .. in` est une *expression*, elle construit un *environnement local* pour *expr*$_2$.

1.3 Commenter ses définitions

Syntaxe: `(*` *text* `*)`

```
(* val aire_cyclindre : float -> float -> float
   (aire_cyclindre r h) renvoie l'aire d'un cylindre
   de rayon r et de hauteur h                        *)
let aire_cyclindre r h =
  (* Constante pi (valeur approximative) *)
  let pi = 3.1416 in
    pi *. r**2. *. h
;;
```

1.4 Les booléens

Type : `bool`

Deux constantes : `true` et `false`

Opérateurs

`not`	négation		
`&&`	conjonction	`&`	synonyme
`||`	disjonctions	`or`	synonyme

Relations

`=`	égalité structurelle	`<>`	négation de `=`
`<`	inférieur	`>=`	supérieur ou égal
`>`	supérieur	`<=`	inférieur ou égal

Typage opérateurs *polymorphes* : on peut comparer soit des entiers, soit des flottants, soit des booléens, etc.

```
# 0 < 1 ;;
- : bool = true
# 0. < 1. ;;
- : bool = true
# false < true ;;
- : bool = true
```

Mais attention, opérateurs *homogènes* : les valeurs comparées ont le même type.

```
# 0 < true ;;
This expression has type int but is here used with type bool
# 0 < 1. ;;
This expression has type float but is here used with type int
```

1.5 Les caractères

Type : `char` Notations : `'A' '\n' '\010'` (octale)

Code ASCII, ISO 8859-1 standard.

```
# int_of_char 'A' ;;
- : int = 10
# char_of_int 65 ;;
- : char = 'A'
# char_of_int 0 ;;
- : char = '\000'
```

Fonction partielle : exception `Invalid_argument`

```
# char_of_int (-1) ;;
Uncaught exception: Invalid_argument "char_of_int".
# char_of_int 256 ;;
Uncaught exception: Invalid_argument "char_of_int".
```

1.6 Chaînes de caractères

Type : `string`

Notation : `"Hello"` `""` (chaîne vide)

Concaténation (opérateur infixe) :

```
# "Hello" ^ " (O'Caml) " ^ "world\n" ;;
- : string = "Hello (O'Caml) world\n"
```

Typage `char` $\neq$ `string`$\neq$ `int` , fonctions de conversion :

```
# 'H' ^ "ello" ;;
This expression has type char but is here used with type string
# "2000" + 1 ;;
This expression has type string but is here used with type int
# int_of_string ;;
- : string -> int = <fun>
# (int_of_string "2000") + 1 ;;
- : int = 2001
```

Fonction partielle : exception `Failure`

```
# int_of_string "2 000" ;;
Uncaught exception: Failure "int_of_string".
```

1.7 Bibliothèque

D'autres fonctions sont fournies dans des *modules*

Syntaxe: `Modname.funame`

Attention : initiale majuscule *obligatoire*

Exemples, modules `Char`, `String`

```
# Char.escaped ;;
- : char -> string = <fun>
# (Char.escaped 'a') ;;
- : string = "a"
```

Recommandation : lire la doc (Part IV)

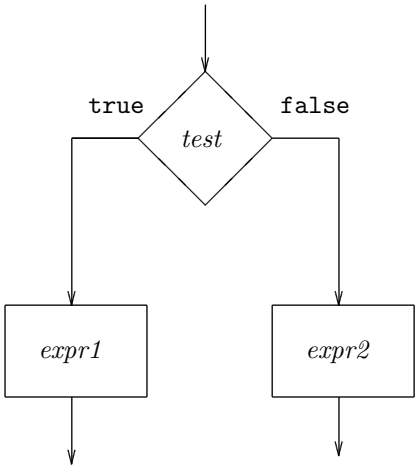
```
val length : string -> int
    Return the length (number of characters) of the given string.

    # (String.length "12345") ;;
    - : int = 5
```

etc.

1.8 Expressions conditionnelles

Syntaxe: `if` *test* `then` *expr1* `else` *expr2*

Typage
– *test* : de type `bool` ;
– *expr1* et *expr2* : type quelconque mais identique (polymorphisme)
Structure de contrôle de l'évaluation : aiguillage.



Exemple : valeur absolue, fonction `abs : int -> int`

```
let abs n =
  if (n < 0) then
    -n
  else
    n
;;
```

1.9 Définitions récursives

Syntaxe: `let rec` *fname* $arg_1 \ldots arg_k$ `=` *expr*

Exemple, la fonction factorielle : $n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$

```
let rec fac n =
  if (n = 0) then
    1
  else
    n * (fac (n - 1))
```

Récursion terminale Appel récursif externe (utile pour optimiser les temps de calculs).

On rajoute un argument qui contiendra le résultat (*accumulateur*)

```
let rec fac_loop n r =
  if (n = 0) then
    r
  else
    (fac_loop (n - 1) (n * r))

let fac n =
    (fac_loop n 1)
```

Utiliser une définition locale : restreindre et contrôler l'utilisation `fac_loop`

```
let fac n =
  let rec loop n r =
    if (n = 0) then
      r
    else
      loop (n - 1) (n * r)
  in
    (loop n 1)
```

Noter : le `n` dans `loop` et `n` dans `fac`, *même nom mais pas même variable.*

1.10 Signaler les erreurs

Attention : `(fac (-1))` *bouclera*. Type $\neq$ domaine de définition. La fonction `fac` est *partielle* sur le type `int` (entiers relatifs).

Utiliser, *déclencher*, les *exceptions*

`val invalid_arg : string -> 'a`
Raise exception `Invalid_argument` *with the given string.*

Tester l'argument *avant* d'appeler la fonction récursive qui fait le calcul

```
let fac n =
  let rec loop n r =
    if (n = 0) then
      r
    else
      (loop (n - 1) (n * r))
  in
    if (n < 0) then
      (invalig_arg "fac")
    else
      (loop n 1)
```

Exemple :

```
# fac (-1) ;;
Exception: Invalid_argument "fac".
```

Noter : parenthèse autour de `-1`

Autre possibilité :

`type exn`
The type of exception values.

`val raise : exn -> 'a`
Raise the given exception value

Noter : résultat polymorphe, "fonction" utilisable partout.

Déclarer ses propres exceptions

```
exception Fac_error of int

let fac n =
  let rec loop n r =
    if (n = 0) then
      r
    else
      loop (n - 1) (n * r)
  in
    if (n < 0) then
      (raise (Fac_error n))
    else
      (loop n 1)
```

Remarque : initiale majuscule *obligatoire*

Exemple :

```
# fac (-1) ;;
Exception: Fac_error -1.
```

1.11 Programme, compilation

On n'utilise pas la boucle d'interaction pour exécuter les fonctions, mais on *compile* un programme.

Il faut gérer les entrées/sorties : écrire un programme qui lit un entier au clavier et affiche sa factorielle.

Primitive d'entrées/sorties standard (clavier/écran)

`val print_int : int -> unit`

Print an integer, in decimal, on standard output.

`val read_int : unit -> int`
Flush standard output, then read one line from standard input and convert it to an integer. Raise `Failure "int_of_string"` *if the line read is not a valid representation of an integer.*

Type `unit` : utilisé quand on se fiche du résultat ou de l'argument. Contient une (et une seule) valeur notée `()`.

Écrire le texte du programme, *fichier source* `fac.ml`

```
(* Fonction *)
let fac n =
  let rec loop n r =
    if (n = 0) then
      r
    else
      loop (n - 1) (n * r)
  in
    if (n < 0) then
      (invalid_arg "fac")
    else
      (loop n 1)
;;

(* Expression ``principale'' *)
print_int (fac (read_int ()))
```

Compiler le source `fac.ml` pour obtenir *l'exécutable* `fac.exe`

```
bash-2.05b: ocamlc -o fac.exe fac.ml
```

Option `-o` : nommer l'exécutable (`a.out` par défaut).

Exécuter le programme

```
bash-2.05b: ./fac.exe
5
120bash-2.05b:
```

1.12 Faire plus joli

Afficher des messages :

`val print_string : string -> unit`
Print a string on standard output.

`val print_newline : unit -> unit`
Print a newline character on standard output, and flush standard output. This can be used to simulate line buffering of standard output.

```
(* Meme fonction *)
let fac n =
  let rec loop n r =
    if (n = 0) then
      r
    else
      loop (n - 1) (n * r)
  in
```

```
    if (n < 0) then
      (invalid_arg "fac")
    else
      (loop n 1)
;;

(* Expression principale plus compliqu\'ee *)
print_string "Valeur pour n : " ;
let n = fac (read_int ()) in
  print_string "Valeur de la factorielle = " ;
  print_int n ;
  print_newline()
```

Compilation puis exécution

```
bash-2.05b: ocamlc -o fac.exe fac.ml
bash-2.05b: ./fac.exe
Valeur pour n : 5
Valeur de la factorielle = 120
bash-2.05b:
```

Séquence le *point-virgule tout seul* permet d'enchaîner l'évaluation (de gauche à droite) d'une *séquence* d'expressions.

Syntaxe: `` $expr_1$ ; ... ; $expr_k$ ``

Typage : $expr_1$, ..., $expr_k$ peuvent avoir n'importe quel type, mais *avertissement* si $expr_1$, ..., $expr_{k-1}$ ne sont pas `unit` ; le type de toute la séquence est celui de $expr_k$.

Valeur : la valeur de la séquence est celle de $expr_k$.

```
let n = print_string "Valeur pour n : " ; read_int () in
  print_string "Valeur de la factorielle = " ;
  print_int (fac n) ;
  print_newline()
```

Sorties formatées module `Printf`, affichages à la C (*format*).

```
let n = Printf.printf "Valeur pour n : " ; read_int () in
  Printf.printf "Valeur de la factorielle = %d\n" (fac n)
```

1.13 Contrôler les erreurs

Exemple d'erreur avec `fac.exe` :

```
bash-2.05b: ./fac.exe
Valeur pour n : douze
Fatal error: exception Failure("int_of_string")
bash-2.05b:
```

Une exception est déclenchée sur saisie incorrecte. On veut redemander une nouvelle saisie : *rattraper* puis traiter l'exception.

Syntaxe: `try` *expr* `with` ex_1 `->` $expr_1$ `|` ... `|` ex_k `->` $expr_k$

Typage : *expr*, $expr_1$, ..., $expr_k$ du même type ; ex_1, ..., ex_k de type `exn`.

Valeur : soit celle de *expr* si pas d'exception ; soit celle de $expr_i$ si *expr* déclenche ex_i.

Exemple : fonction de lecture d'un entier sûre :

```
let rec safe_read_int () =
  try
    read_int()
  with
    Failure("int_of_string") -> (
      print_string "redo ? ";
      safe_read_int ()
    )
```

Noter : les parenthèses autour de la séquence.

Utilisation :

```
bash-2.05b: ./fac.exe
Valeur pour n : douze
redo ? 12
Valeur de la factorielle = 479001600
bash-2.05b:
```

1.14 Modifier l'état mémoire

Construire une *référence à une valeur*

`val ref : 'a -> 'a ref`
Return a fresh reference containing the given value.

Noter : fonction polymorphe

Exemple : référence à la valeur entière 0

```
# ref 0 ;;
- : int ref = {contents = 0}
```

Noter : la *variable de type* `'a` est devenue `int`, elle a été *instanciée*.

Nommer une référence, comme d'hab'

```
# let n = ref 0 ;;
val n : int ref = {contents = 0}
```

Modifier la valeur référencée

`val ( :=) : 'a ref -> 'a -> unit`
`r := a` *stores the value of* `a` *in reference* `r`.

Exemple

```
# n := 100 ;;
- : unit = ()
```

Accéder à la valeur référencée

`val (!) : 'a ref -> 'a`
`!r` *returns the current contents of reference* `r`.

Exemple

```
# !n ;;
- : int = 100
```

Contrôle (:=) expression de droite évaluée *avant* la modification de la référence à gauche.

Exemple : modifier la valeur de `n` en fonction de `n`, incrémenter de 1 :

```
# n := !n + 1 ;;
- : unit = ()
```

Regarder alors ce que *contient* `n`

```
# !n ;;
- : int = 101
```

1.15 Fonction *vs* procédure

Séquence *vs* composition (l'accumulateur est une référence)

```
let fact n =
  let rec loop n r =
    if n = 0 then
      !r
    else (
      r := n * !r;
      loop (n - 1) r
    )
  in
    loop n (ref 1)
```

Modifier l'environnement local : l'accumulateur est dans l'environnement local.

```
let fac n =
  let r = ref 1 in
  let rec loop n =
    if (n = 0) then
      !r
    else (
      r := n * !r;
      loop (n - 1)
    )
  in
    loop n
```

1.16 Itération *vs* récursion

Vider la fonction `loop` de ses arguments, inverser la condition et ne rien faire quand on atteind `0`, le résultat est dans `r` :

```
let fac n =
  let r = ref 1 in
  let i = ref n in
  (* val loop: unit -> unit *)
  let rec loop () =
    if (!i <> 0) then (
      r := !i * !r;
      i := !i - 1;
      loop ()
    )
```

```
in
  loop ();
  !r
```

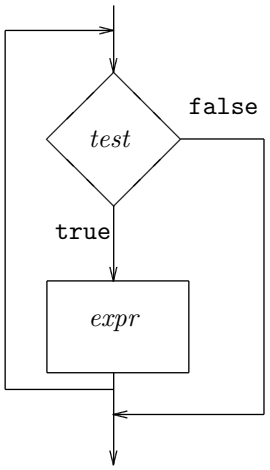
Noter : le `if` *unilatère* (sucre syntaxique pour `unit` seulement)

`if` *test* `then` *expr* ≡ `if` *test* `then` *expr* `else ()`

La boucle structure de contrôle de l'itération

Syntaxe: `while` *test* `do` *expr* `done`

Contrôle



Typage
– *test* : type `bool` ;
– *expr* : type quelconque, mais généralement `unit` (avertissement sinon)
Factorielle itérative

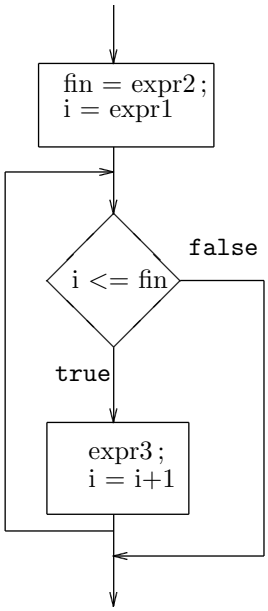
```
let fac n =
  let r = ref 1 in
  let i = ref n in
    while (!i <> 0) do
      r := !i * !r;
      i := !i - 1
    done;
    !r
```

1.17 Itération bornée

Syntaxe: `for` *vname* `=` $expr_1$ `to` $expr_2$ `do` $expr_3$ `done`

Typage
– $expr_1$ et $expr_2$: type `int`
– $expr_3$: type quelconque, mais plutôt `unit`

Contrôle



Noter : la valeur de fin est calculée en premier (*fin*), la sortie de boucle ne dépend pas de $expr_3$.

Exemple

```
let fac n =
  let r = ref 1 in
    for i = 2 to n do
      r := !r * i
    done;
    !r
```

Noter : `(fac 0)`=`(fac 1)`=`1`, pas de boucle.

Variante indice décroissant, mot clef `downto`

```
let fac n =
  let r = ref 1 in
    for i = n downto 2 do
      r := !r * i
    done;
    !r
```

1.18 Fonctionnelle d'itération

Problème : compter le nombre d'occurences d'un caractère `c` dans une chaîne `s`.

Une solution avec un `for`

```
let char_count s c =
  let r = ref 0 in
    for i = 0 to (String.length s) - 1 do
      if s.[i] = c then incr r
    done;
```

```
    !r
```

Une autre solution utilisant un *itérateur* (module `String`)

`val iter : (char -> unit) -> string -> unit`
`(String.iter f s)` *applies function* `f` *in turn to all the characters of* `s`*. It is equivalent to* `f s.(0); f s.(1); ...; f s.(String.length s - 1); ()`.

On écrit une fonction locale chargée de réaliser l'incrément conditionnel que l'on avait dans le corps de la boucle.

```
let char_count s c =
  let r = ref 0 in
  let f s_i =
    if s_i = c then incr r
  in
    String.iter f s;
    !r
```

Expression fonctionnelle fonction anonyme

```
let char_count s c =
  let r = ref 0 in
    String.iter (fun s_i -> if s_i = c then incr r) s;
    !r
```

Syntaxe: `fun` $arg_1 \ldots arg_n$ `->` *expr*

2 Entrées/sorties

2.1 Canaux de communication

Deux types :

`type in_channel`
The type of input channel.

`out_channel`
The type of output channel.

Ouverture/création des canaux : liaison avec les noms de fichiers

`val open_in : string -> in_channel`
Open the named file for reading, and return a new input channel on that file, positionned at the beginning of the file. Raise `Sys_error` *if the file could not be opened.*

`val open_out : string -> out_channel`
Open the named file for writing, and return a new output channel on that file, positionned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists. Raise `Sys_error` *if the file could not be opened.*

Fermetures des canaux

`val close_in : in_channel -> unit`
Close the given channel. Input functions raise a `Sys_error` *exception when they are applied to a closed input channel, except* `close_in`*, which does nothing when applied to an already closed channel.*

`val close_out : out_channel -> unit`
Close the given channel, flushing all buffered write operations. Output functions raise a `Sys_error` *exception when they are applied to a closed output channel, except* `close_out` *and* `flush`*, which do nothing when applied to an already closed channel.*

2.2 Écrire dans un fichier

Fichier texte : lignes, chaînes de caractères avec retour chariot.

Fonction d'écriture

`val output_string : out_channel -> string -> unit`
Write the string on the given output channel.

Forçage des écritures (*temporisées*).

`val flush : out_channel -> unit`
Flush the buffer associated with the given output channel, performing all pending writes on that channel. Interactive programs must be careful about flushing standard output and standard error at the right time.

Exemple, affichage (écriture) sur la sortie standard (écran)

```
let print_msg s =
  output_string stdout s;
  flush stdout
```

Auxiliaire : saisie d'informations ; nom, prénom et numéro de téléphone ; ajout séparateur (caractère `:`), fin de ligne (caractère `\n`)

```
let get_info () =
  let nom = print_msg "Nom: "; read_line ()
```

```
  and prenom = print_msg "Prenom: "; read_line ()
  and num =  print_msg "Numero: "; read_line () in
    nom ^ ":" ^ prenom ^ ":" ^ num ^ "\n"
```

Confirmation saisie

```
let rec get_info_ok () =
  let info = get_info () in
    print_msg "Ok (o/[n]) ? ";
    if (read_line () = "o") then
      info
    else
      get_info_ok ()
```

Saisie et écriture dans un fichier (canal de sortie) ; s'arrête quand saisie vide.

```
let rec get_and_write_info oc =
  let info = get_info_ok () in
    if (info  = "::\n") then (
      print_msg "Confirmer sortie (o/[n]) ? ";
      if (read_line () <> "o") then
get_and_write_info oc
    )
    else (
      output_string oc info;
      flush_out oc;
      get_and_write_info oc
    )
```

Création fichier et écriture

```
let create_info_file fname =
  let oc = open_out fname in
    get_and_write_info oc;
    close_out oc
```

Ajout d'information (en fin de fichier), autre primitive de création de canal.

`val open_out_gen : open_flag list -> int -> string -> out_channel`
Open the named file for writing, as above. The extra argument mode specify the opening mode. The extra argument perm specifies the file permissions, in case the file must be created. `open_out` *and* `open_out_bin` *are special cases of this function.*

Type `list` voir plus tard, premissions, voir cours système.

`type open_flag = .. | Open_wronly | Open_append | ..`
Opening modes for open_out_gen and open_in_gen.
...
Open_wronly : open for writing.
Open_append : open for appending.
...

```
let add_info_file fname =
  let oc = open_out_gen [Open_append] 0 fname in
    get_and_write_info oc;
    close_out oc
```

2.3 Lecture d'un fichier

Fonction de lecture d'une ligne

`val input_line : in_channel -> string`
Read characters from the given input channel, until a newline character is encountered. Return the string of all characters read, without the newline character at the end. Raise `End_of_file` *if the end of the file is reached at the beginning of line.*

Utiliser l'exception pour lire jusqu'à la fin de fichier :

```
let cat_info_file fname =
  let ic = open_in fname in
    try
      while true do
        print_endline (input_line ic)
      done
    with
      End_of_file -> close_in ic
```

Position dans un fichier

`val seek_out : out_channel -> int -> unit`
`(seek_out chan pos)` *sets the current writing position to* `pos` *for channel* `chan`*. This works only for regular files. On files of other kinds (such as terminals, pipes and sockets), the behavior is unspecified.*

`val pos_out : out_channel -> int`
Return the current writing position for the given channel.

`val seek_in : in_channel -> int -> unit`
`(seek_in chan pos)` *sets the current reading position to* `pos` *for channel* `chan`*. This works only for regular files. On files of other kinds, the behavior is unspecified.*

`val pos_in : in_channel -> int`
Return the current reading position for the given channel.

Exemple, recherche et marquage d'une ligne d'info

```
let search_info_pos fname =
  let ic = open_in fname in
  let nm = get_field "Nom: " in
  let i = ref 0 in
  let rec loop () =
    let info = input_line ic in
      if (comp_nom nm info) then (
close_in ic;
        info, !i
      )
      else (
        i := pos_in ic;
        loop ()
      )
  in
    try loop ()
    with End_of_file -> (close_in ic; raise Not_found)

let del_info_file fname =
  let info, pos = search_info_pos fname in
  let oc = open_out_gen [Open_wronly] 0 fname in
    Printf.printf"%d: %s\n" pos info;
    seek_out oc pos;
    output_char oc '*';
    close_out oc
```

2.4 Éléments d'analyse lexicale

(Tenter de) lire un caractère statisfaisant un certain critère, sinon ne pas bouger et signaler par une exception.

```
exception No_reading

let read_c ic is_ok =
  let c = input_char ic in
    if (is_ok c) then
      c
    else (
      seek_in ic (pred (pos_in ic));
      raise No_reading
    )
```

Lire une suite de caractères statisfaisant un certain critère. Peut déclencher `No_reading` ou `End_of_file` si rien n'a pu être lu.

```
let read_s ic is_ok =
  let i = ref 0 in
  let buff = String.create 1024 in
  let rec loop () =
    try
      let c = read_c ic is_ok in
        buff.[!i] <- c;
        incr i;
        if !i < 1024 then
          loop()
        else
          failwith "read: buffer overflow"
     with
       e -> (* expects End_of_file or No_reading *)
         if (!i = 0) then
           raise e
         else
           String.sub buff 0 !i
  in
    loop ()
```

Application : lire une suite de mots (alpha-numèriques) séparés par des espaces, les afficher avec leur numéro.

```
let is_sep c = (c = ' ')

let is_alpha_num c =
  (('a' <= c) && (c <= 'z')) or (('A' <= c) && (c <= 'Z'))
  or (('0' <= c) && (c <= '9'))

let read_words fname =
  let ic = open_in fname in
  let i = ref 1 in
  let rec loop () =
    let w = read_s ic is_alpha_num in
      Printf.printf "%d: %s\n" !i w;
      incr i;
      ignore(read_s ic is_sep);
```

```
      loop ()
  in
    try loop () with _ -> close_in ic
```

Noter utilisation de `ignore :'a -> unit`.

Essai sur le fichier `words_line.txt` contenant la ligne :

```
This is  a line  of words  separated  by 1 or 2 or  3   spaces
```

```
# read_words "words_line.txt";;
1: This
2: is
3: a
4: line
5: of
6: words
7: separated
8: by
9: 1
10: or
11: 2
12: or
13: 3
14: spaces
- : unit = ()
```

Exo : additionner les entiers contenus sur une ligne de fichier

Autres type de fichiers voir 3.3

3 Structures de données

3.1 Les n-uplets

Équivalent du produit cartésien ensembliste.

Les paires : donnée encapsulant deux valeurs.

Notation du type : `*`

Exemples : `int * int`, `int * string`, `'a * 'b`

Contruction d'une valeur : `( , )`

Exemples :

```
# (23, 45) ;;
- : int * int = (23, 45)
# (12, "december") ;;
- : int * string = (12, "december")
```

Calculer deux valeurs = calculer *un couple* de valeurs : couper une chaîne en deux selon un caractère de séparation.

```
# let split_string s c =
    let i = String.index s c in
    let i' = i + 1 in
      ((String.sub s 0 i), (String.sub s i' ((String.length s) - i')))
;;
val split_string : string -> char -> string * string = <fun>
# split_string "1234 5678" ' ';;
- : string * string = ("1234", "5678")
```

Avec (module `String`)

`val index : string -> char -> int`
`(String.index s c)` *returns the position of the leftmost occurrence of character* `c` *in string* `s`*.*
Raise `Not_found` *if* `c` *does not occur in* `s`*.*

Accesseurs (core library)

`val fst : 'a * 'b -> 'a`
Return the first component of a pair.

`val snd : 'a * 'b -> 'b`
Return the second component of a pair.

Les triplets : donnée encapsulant trois valeurs.

Notation du type : `* *`

Exemples : `int * int * float`, `int * string * string`, `'a * 'b * 'c`

Contruction d'une valeur : `( , , )`

Exemples :

```
# (23, 45, 67.89) ;;
- : int * int * float = (23, 45, 67.89)
# (12, "december", "dec.") ;;
- : int * string * string = (12, "december", "dec.")
```

Pas de fonction d'accés standard, les définir soit même

```
let thrd (x,y,z) = z
```

Attention :

```
# (fst (1, 2, 3)) ;;
This expression has type int * int * int but is here used with type 'a * 'b
```

Etc. les n-uplets

3.2 Les enregistrements

Comme des n-uplets, mais les composants sont nommés. On les appelle des *champs*.

Exemple structure pour représenter les nombres rationnels (ensemble $\mathbb{Q}$)

Définition du type :

```
# type ratio = { num:int; den: int } ;;
```

Syntaxe: `type` [*tparams*] *tname* `=` { *fname*$_1$ `:`*typexp*$_1$ `;` ... `;` *fname*$_k$ `:`*typexp*$_k$ }

Construction d'une valeur :

```
# let q = { num = 3; den = 4 } ;;
val q : ratio = {num = 3; den = 4}
```

Syntaxe: { *fname*$_1$`=`*expr*$_1$ `;` ... `;` *fname*$_k$`=`*expr*$_k$ }

Accés aux éléments de la structure : par le nom des champs, notation pointée

```
# q.num ;;
- : int = 3
# q.den ;;
- : int = 4
```

Syntaxe: *expr* `.` *fname*

Champs modifiables mot clef `mutable`. La vérité sur le type `ref` (core library)

```
type 'a ref = { mutable contents : 'a ; }
```
The type of references (mutable indirection cells) containing a value of type 'a.

Noter : *type paramétré*, polymorphe, *variable* de type (`'a`).

Syntaxe: `type` [*tparams*] *tname* `=` { [`mutable`] *fname*$_1$ `:`*typexp*$_1$ `;` ... `;` [`mutable`] *fname*$_k$ `:`*typexp*$_k$ }

Modification :

```
# let pi = { contents = 3.1416 } ;;
val pi : float ref = {contents = 3.1416}
# pi.contents <- 3.1415926535 ;;
- : unit = ()
# pi ;;
- : float ref = {contents = 3.1415926535}
```

Syntaxe: *expr*$_1$ `.` *fname* `<-` *expr*$_2$

3.3 Les listes

Structures linéaires *dynamiques*. Suites finies de valeurs d'un même type. Type paramétré.

Type : t `list`

Constructeurs :
– `[]`, la liste vide
– `::`, ajoute un élément en tête.
Exemple :

```
# [] ;;
- : 'a list = []
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Noter : l'*instanciation* du paramètre de type.

Sucre syntaxique :

```
# [1; 2; 3] ;;
- : int list = [1; 2; 3]
# 0 :: [1; 2; 3] ;;
- : int list = [0; 1; 2; 3]
```

Listes *homogènes*

```
# true :: [1; 2; 3] ;;
This expression has type int but is here used with type bool
```

Accesseurs (module `List`) pour les listes non vides

`val hd : 'a list -> 'a`
Return the first element of the given list. Raise `Failure "hd"` *if the list is empty.*

`val tl : 'a list -> 'a list`
Return the given list without its first element. Raise `Failure "tl"` *if the list is empty.*

Exemple :

```
# (List.hd [1; 2; 3]) ;;
- : int = 1
# (List.tl [1; 2; 3]) ;;
- : int list = [2; 3]
```

Prédicat, reconnaisseur non fourni

```
let is_empty_list xs = (xs = [])
```

Liste et récurence Définition *récursive* de l'ensemble des listes :

1. `[]` est une liste.
2. si `xs` est une liste et `x` un élément alors `x :: xs` (prononcer :x *conse* xs) est une liste.

Exemple de fonction récursive avec les listes : construire une liste qui double la valeur des éléments de la liste passée en argument.

```
# let rec double ns =
    if (is_empty_list ns) then
      []
    else
      ((List.hd ns) * 2) :: (double (List.tl ns))
;;
val double : int list -> int list = <fun>
# (double [1; 2; 3; 4]) ;;
- : int list = [2; 4; 6; 8]
```

Filtrage de la définition récursive, on tire : toute liste a

1. soit *la forme* `[]`
2. soit *la forme* `debut :: suite`

Pour forme, on dit *motif* (*pattern*)

Filtrage/*pattern matching* : primitive générique d'accés aux éléments d'une structure (ici, les listes)

Syntaxe: `match` *expr* `with [] ->` $expr_1$ `|` pat_1 `::` pat_2 `->` $expr_2$

Nous reviendrons sur le filtrage en 5

Exemple : la fonction `double`

```
# let rec double ns =
    match ns with
      [] -> []
    | n1 :: ns1 -> (n1 * 2) :: (double ns1)
;;
val double : int list -> int list = <fun>
# (double [1; 2; 3; 4]) ;;
- : int list = [2; 4; 6; 8]
```

Autre exemple : position d'un élément dans une liste

```
# let list_index x xs =
  let rec loop xs i =
    match xs with
        [] -> raise Not_found
      | x1::x1s ->
          if (x = x1) then i
          else (loop x1s (succ i))
  in
    (loop xs 0)
                  ;;
val list_index : 'a -> 'a list -> int = <fun>
# (list_index 1 [1;2;3;4;5]) ;;
- : int = 0
# (list_index 3 [1;2;3;4;5]) ;;
- : int = 2
# (list_index 12345 [1;2;3;4;5]) ;;
Exception: Not_found.
```

Fonctionnelle (module `List`)

`val map : ('a -> 'b) -> 'a list -> 'b list`

List.map f [a1 ; ... ; an] applies function f to a1, ..., an, and builds the list [f a1 ; ... ; f an] with the results returned by f. Not tail-recursive.

```
# let double ns =
    List.map (fun n -> n * 2) ns
;;
val double : int list -> int list = <fun>
# (double [1; 2; 3; 4]) ;;
- : int list = [2; 4; 6; 8]
```

`val iter : f :('a -> unit) -> 'a list -> unit`

`(List.iter f [a1 ; ... ; an])` *applies function* `f` *in turn to* `a1` *; ... ;* `an`*. It is equivalent to* `begin (f a1) ; (f a2) ; ... ; (f an) ; () end`*.*

```
# let prints print_x xs =
    List.iter print_x xs ;;
val prints : ('a -> unit) -> 'a list -> unit = <fun>
# let print_int n =
    Printf.printf "(%d)" n ;;
val print_int : int -> unit = <fun>
# prints print_int [1; 2; 3; 4] ;;
(1)(2)(3)(4)- : unit = ()
```

Exos : `find_min` avec `compare`-like function

```
let find_min comp xs =
  let rec loop x xs =
    match xs with
      [] -> x
    | x'::xs ->
      if (comp x' x) < 0 then loop x' xs
      else loop x xs
  in
    match xs with
        [] -> raise Not_found
    | x::xs -> loop x xs
```

Application : le jeu de dominos. `http ://www.momes.net/jeux/pij/jeuxdedominos.html`

Entrées/sorties Sauvegarde des éléments d'une liste, format *texte*, format *binaire* ; ou sauvegarde de la *structure* : encodage (*marshaling*, format binaire).

Format texte ou binaire : fonction d'écriture générique (paramètre : fonction d'écriture d'un élément)

```
let rec write_list oc write_x xs =
  List.iter (write_x oc) xs
```

Utilisation 1, liste d'entier format binaire

`val output_binary_int : out_channel -> int -> unit`

Write one integer in binary format on the given output channel. The only reliable way to read it back is through the `input_binary_int` *function. The format is compatible across all machines for a given version of Objective Caml.*

```
# let oc = open_out "lbin.dat";;
write_list oc output_binary_int [123;456;789];
close_out oc;;
val oc : out_channel = <abstr>
#   - : unit = ()
```

on obtient

```
bash-2.05b: file lbin.dat
lbin.dat: data
bash-2.05b: more lbin.dat
^@^@^@{^@^@^A<C8>^@^@^C^U
```

Utilisation 2, liste d'entiers au format texte (conversion)

```
# let oc = open_out "ltxt.txt";;
let write_as_txt oc n =
  output_string oc (string_of_int n);
  output_char oc '\n'
in
  write_list oc write_as_txt [123;456;789];
  close_out oc;;
val oc : out_channel = <abstr>
#   - : unit = ()
```

on obtient

```
bash-2.05b: file ltxt.txt
ltxt.txt: ASCII text
bash-2.05b: more ltxt.txt
123
456
789
```

Structure encodée

`val output_value : out_channel -> 'a -> unit`

Write the representation of a structured value of any type to a channel. Circularities and sharing inside the value are detected and preserved. The object can be read back, by the function `input_value`*. See the description of module* `Marshal` *for more information. [...]*

```
# let oc = open_out "lvalue.dat" ;;
output_value oc [123; 456; 789];
close_out oc ;;
```

on obtient

```
bash-2.05b: file lvalue.dat
lvalue.dat: data
bash-2.05b: more lvalue.dat
<84><95><A6><BE>^@^@^@^L^@^@^@^C^@^@^@  ^@^@^@  <A0>^@{<A0>^A^A<C8>
<A0>^A^C^U@
```

Lecture des éléments d'une liste dans un fichier, paramétré par la fonction de lecture d'un élément.

```
let rec read_list ic read_x =
  let rec loop () =
    try
      let x = read_x ic in
x::(loop ())
    with
End_of_file -> []
  in
    loop ()
```

Exemple 1, lecture d'un liste d'entier à partir d'un *fichier binaire*

```
# let ic = open_in "lbin.dat" ;;
val ic : in_channel = <abstr>
# read_list ic input_binary_int ;;
- : int list = [123; 456; 789]
# close_in ic ;;
- : unit = ()
```

Exemple 2, lecture d'un liste d'entier à partir d'un *fichier texte* (un élément par ligne)

```
# let ic = open_in "ltxt.txt" ;;
val ic : in_channel = <abstr>
# read_list ic input_line ;;
- : string list = ["123"; "456"; "789"]
# close_in ic ;;
- : unit = ()
```

Attention au mélange des genres :

```
# let ic = open_in "ltxt.txt" ;;
val ic : in_channel = <abstr>
# read_list ic input_binary_int ;;
- : int list = [825373450; 875902474; 926431498]
# close_in ic ;;
- : unit = ()
```

Lecture d'une structure encodée :

```
# input_value ;;
- : in_channel -> 'a = <fun>
# let ic = open_in "lvalue.dat" ;;
val ic : in_channel = <abstr>
# let ns = input_value ic ;;
val ns : '_a = <poly>
```

Fonction polymorphe (résultat indéterminé) : noter la variable *de type faible* `'_a` ; mais connue (du programmeur) :

```
# prints print_int ns ;;
(123)(456)(789)- : unit = ()
```

Le type faible est désormais connu :

```
# ns ;;
- : int list = [123; 456; 789]
```

Containte de type

Syntaxe: (*expr* : *typexp*)

utilisation

```
# let ic = open_in "lvalue.dat" ;;
val ic : in_channel = <abstr>
# (input_value ic : int list) ;;
- : int list = [123; 456; 789]
# close_in ic ;;
- : unit = ()
```

Exos : avec `input_byte`.

3.4 Tableaux

Structures linéaires *statique*. Suites finies d'éléments de même type. Type paramétré.

Type : *typexp* `array`

Constructeur création d'un tableau, module `Array`

```
val make : int -> 'a -> 'a array
```

`(Array.make n x)` *returns a fresh array of length* `n`*, initialized with* `x`*. All the elements of this new array are initially physically equal to* `x` *(in the sense of the* `==` *predicate). Consequently, if* `x` *is mutable, it is shared among all elements of the array, and modifying* `x` *through one of the array entries will modify all other entries at the same time.*
Raise `Invalid_argument` *if* $n < 0$ *or* $n >$ `Sys.max_array_length`*. If the value of* `x` *is a floating-point number, then the maximum size is only* `Sys.max_array_length / 2`*.*

```
# let a = Array.make 4 0 ;;
val a : int array = [|0; 0; 0; 0|]
```

Accesseur suite d'élément numérotés (*indicés*).

`val get : 'a array -> int -> 'a`
`(Array.get a n)` *returns the element number* `n` *of array* `a`*. The first element has number 0. The last element has number* `(Array.length a) - 1`*.*
Raise `Invalid_argument "Array.get"` *if* `n` *is outside the range 0 to* `(Array.length a) - 1`*. You can also write* `a.(n)` *instead of* `(Array.get a n)`*.*

```
# a.(0) ;;
- : int = 0
# a.(4) ;;
Exception: Invalid_argument "Array.get".
```

Donnée modifiable

`val set : 'a array -> int -> 'a -> unit`
`(Array.set a n x)` *modifies array* `a` *in place, replacing element number* `n` *with* `x`*.*
Raise `Invalid_argument "Array.set"` *if* `n` *is outside the range 0 to* `(Array.length a) - 1`*. You can also write* `a.(n) <- x` *instead of* `(Array.set a n x)`*.*

```
# a.(0) <- 1; a.(1) <- 2; a.(2) <- 3; a.(3) <- 4 ;;
- : unit = ()
# a ;;
- : int array = [|1; 2; 3; 4|]
```

Tableau et itération bornée la manière standard de manipuler les tableaux est l'itération bornée.

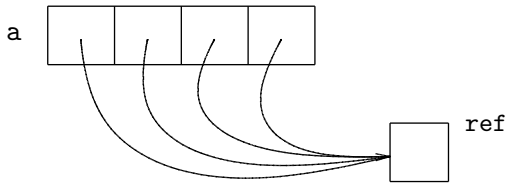
```
# let double a =
    for i = 0 to (Array.length a) - 1 do
      a.(i) <- a.(i) * 2
    done ;;
val double : int array -> unit = <fun>
# double a ;;
- : unit = ()
# a ;;
- : int array = [|2; 4; 6; 8|]
```

Modèle mémoire Retour sur : *"All the elements of this new array are initially physically equal to* `x` *(in the sense of the* `==` *predicate). Consequently, if* `x` *is mutable, it is shared among all elements of the array, and modifying* `x` *through one of the array entries will modify all other entries at the same time."*

Attention :

```
# let a = Array.make 4 (ref 0) ;;
val a : int ref array =
```

```
  [|{contents = 0}; {contents = 0}; {contents = 0}; {contents = 0}|]
# incr a.(0) ;;
- : unit = ()
# a ;;
- : int ref array =
[|{contents = 1}; {contents = 1}; {contents = 1}; {contents = 1}|]
```



Tous les éléments du tableau *partagent* la même référence.

Égalité physique savoir si deux objets sont *le même* objet mémoire

Syntaxe: $expr_1$ `==` $expr_2$

Exemple

```
# a.(0) == a.(1) ;;
- : bool = true
```

Remède : une référence propre à chaque case du tableau

```
# let make_ref_array n x =
    let a = Array.make n (ref x) in
      for i=0 to n-1 do a.(i) <- (ref x) done;
      a
;;
val make_ref_array : int -> 'a -> 'a ref array = <fun>
# let a = make_ref_array 4 0 ;;
val a : int ref array =
  [|{contents = 0}; {contents = 0}; {contents = 0}; {contents = 0}|]
# incr a.(0) ;;
- : unit = ()
# a ;;
- : int ref array =
[|{contents = 1}; {contents = 0}; {contents = 0}; {contents = 0}|]
```

On a bien supprimé le *partage*

```
# a.(0) == a.(1) ;;
- : bool = false
```

(Exos : matrices)

4 Types abstraits, modules simples

Modéliser un jeu de cartes : couleurs, figures, cartes et paquet.

4.1 Module

Les couleurs d'un jeu de cartes : 4 couleurs = 4 entiers + une couleur "nulle" + conversion.

```
let color_null = -1

let color_coeur = 0
let color_carreau = 1
let color_pique = 2
let color_trefle = 3

let color_strings = [| "Coeur"; "Carreau"; "Pique"; "Trefle" |]

let color_to_string c = color_strings.(c)
```

Idée : en faire un *module* (simple) ; fichier regroupant des définitions (types, valeurs, exceptions, ..)

Exemple : fichier `color.ml`

```
let null = -1

let coeur = 0
let carreau = 1
let pique = 2
let trefle = 3

let strings = [| "Coeur"; "Carreau"; "Pique"; "Trefle" |]

let to_string c = strings.(c)
```

Noter : le préfixe `color_` a disparu.

Compilation du module :

```
bash-2.05b: ocamlc -c -i color.ml
val null : int
val coeur : int
val carreau : int
val pique : int
val trefle : int
val strings : string array
val to_string : int -> string
bash-2.05b: ls color.*
color.cmi       color.cmo       color.ml
```

Options :
`-c` pas d'édition de lien, code *objet*, pas exécutable
`-i` pour voir les types
Fichiers engendrés
`.cmo` le code objet
`.cmi` l'*interface*, la *signature*

Signature : ensemble de déclaration (nom+type) – cf affichage

Utilisation du module (au *toplevel*)

```
# #load "color.cmo";;
# Color.coeur ;;
- : int = 0
# Color.to_string Color.coeur ;;
- : string = "Coeur"
```

Problème :

```
# Color.to_string ;;
- : int -> string = <fun>
# Color.to_string 12345 ;;
Exception: Invalid_argument "Array.get".
```

On voudrait : `Color.to_string : color -> string`, un *type* pour les couleurs.

On va faire `Color.to_string : Color.t -> string`

4.2 Type abstrait

Idée : on représente toujours les couleurs par des entiers, mais *on le cache*.

On *déclare* (sans le *définir*) un type `Color.t` avec valeurs et fonctions associées. Fichier d'interface explicite (source) : `color.mli`

```
type t

val null : t
val coeur : t
val carreau : t
val pique : t
val trefle : t
val to_string : t -> string
```

Noter :
– pas de préfixe au type `t`
– on a aussi caché le tableau `strings`
Compilation :

```
bash-2.05b: ocamlc color.mli
```

pour obtenir le `.cmi` voulu.

On *définit* l'*implanation* de la signature (fichier `color.ml`)

```
type t = int

let null = -1

let coeur = 0
let carreau = 1
let pique = 2
let trefle = 3

let strings = [| "Coeur"; "Carreau"; "Pique"; "Trefle" |]

let to_string c = strings.(c)
```

On a rajouté la définition du type `t`

Compilation :

```
bash-2.05b: ocamlc -c color.ml
```

Utilisation du module (au *toplevel*)

```
# #load "color.cmo";;
# Color.coeur ;;
- : Color.t = <abstr>
```

On a obtenu un *type abstrait*.

```
# Color.to_string ;;
- : Color.t -> string = <fun>
# Color.to_string Color.coeur ;;
- : string = "Coeur"
# Color.to_string 12345 ;;
This expression has type int but is here used with type Color.t
```

Figure idem

Signature (`fig.mli`) :

```
type t

val null : t

val ace : t
val dame : t
val dix : t
val huit : t
val neuf : t
val roi : t
val sept : t
val valet : t

val to_string : t -> string
```

Implanation (`fig.ml`) :

```
type t = int

let null = -1

let ace = 0
let dame = 1
let dix = 2
let huit = 3
let neuf = 4
let roi = 5
let sept = 6
let valet = 7

let strings = [| "As"; "Dame"; "Dix"; "Huit"; "Neuf"; "Roi"; "Sept"; "Valet" |]

let to_string f = strings.(f)
```

Cartes type abstrait

Signature (commentée), fichier `card.mli` :

```
(** Module: Card *)

type t
(** Abstract type for cards *)

val null : t
(** The ``null'' card (for default value) *)

val make : Fig.t -> Color.t -> t
(** [(Card.make f c)] create a card with figure [f] and color [c] *)

val get_col : t -> Color.t
(** Return the color of the given card *)

val get_fig : t -> Fig.t
(** Return the figure of the given card *)

val to_string : t -> string
(** Return a string naming (in french) the given card *)
```

`null` et `make` sont les *constructeurs* du type abstrait `Card.t`

Implantation, fichier `card.ml` :

```
type t = Fig.t * Color.t

let null = (Fig.null, Color.null)
let make f c = (f,c)

let get_col = snd
let get_fig = fst

let to_string c =
  if (c = null) then
    "[]"
  else
    Printf.sprintf "[%s de %s]"
                   (Fig.to_string (get_fig c))
                   (Color.to_string (get_col c))

let print c = print_string (to_string c)
```

Compilation (signature et implanation :

```
bash-2.05b: ocamlc -c card.mli card.ml
bash-2.05b: ls card.*
card.cmi        card.cmo        card.ml         card.mli
```

Utilisation (au *toplevel*) : charger *tous* les module nécessaires

```
# #load"color.cmo";;
# #load"fig.cmo";;
# #load"card.cmo";;
```

Application : valeur d'une carte (à la belote, fonction de la couleur d'atout)

```
# let get_value ca c =
  let co = (Card.get_col c) in
  let fg = (Card.get_fig c) in
    if (fg = Fig.valet) then
      if (co = ca) then 20 else 2
    else if (fg = Fig.neuf) then
      if (co = ca) then 14 else 0
    else if (fg = Fig.ace) then 11
    else if (fg = Fig.dix) then 10
    else if (fg = Fig.roi) then 4
    else if (fg = Fig.dame) then 3
    else 0
                          ;;
val get_value : Color.t -> Card.t -> int = <fun>
```

Application : trouver des combinaisons (brelan, carré, suites, ..)

5 Types algébriques, types somme ou variants

Type couleur : type *énuméré* ; ensemble finis de *constantes*.

Idée : un nom pour chaque constante.

Réalisé par le type abstrait/module `Color` : on connaît les noms (`Color.coeur`, `Color.pique`, etc.) mais pas leur valeur *concrète* (masquage pas signature).

Possibilité en ML : définition directe d'un type algébrique, *types sommes* ou *variants* ($\neq$ types *produit* : n-uplets, enregistrement).

Exemples couleur et figures :

```
type color = Coeur | Carreau | Trefle | Pique

type fig = As | Dame | Dix | Huit | Neuf | Roi | Sept | Valet
```

Constantes (nom de) : *constructeurs* du type.

Syntaxe : majuscules *obligatoires*

```
# type color = coeur | carreau | trefle | pique ;;
Syntax error
```

Constantes = *valeurs* de type `color` et `fig`

```
# As ;;
# Coeur ;;
- : color = Coeur
# (As, Coeur) ;;
- : fig * color = (As, Coeur)
```

Filtrage : programmation par cas, mots clefs : `match...with`

```
# let string_of_color c =
    match c with
      Coeur -> "coeur"
    | Carreau -> "carreau"
    | Trefle -> "trefle"
    | Pique -> "pique"
val string_of_color : color -> string = <fun>
```

À gauche de `->` : *motifs* (de filtrage) ; *patterns*

Filtrage plus complexe : couples (Constantes + variables) ; motif *universel*

```
# let get_values ca c =
  match c with
      (Valet, co) -> if (ca = co) then 20 else 2
    | (Neuf, co) -> if (ca = co) then 14 else 0
    | (As, _) -> 11
    | (Dix, _) -> 10
    | (Roi, _) -> 4
    | (Dame, _) -> 3
    | _ -> 0
;;
val get_values : 'a -> fig * 'a -> int = <fun>
```

Remarquer : type *inféré polymorphe* ; la (définition de le) fonction n'utilise pas et ne donne pas d'information sur le type `ca` ou `co` (égalité elle-même polymorphe).

Constructeurs fonctionnels pour avoir une "carte nulle", mot clef `of`

```
type t = Null | Make of fig * color
```

Interprétation : une carte (élément du type `t`) est
ou bien la carte nulle, notée `Null` ;
ou bien la carte construite à partir d'une figure `f` et d'une couleur `t`, notée `Make(f,c)`

```
# Null;;
- : t = Null
# Make(As, Coeur) ;;
- : t = Make (As, Coeur)
```

Syntaxe: `type` *tname* `=` *Cname*$_1$ [`of` *typexp*$_1$] `|` ... `|` *Cname*$_k$ [`of` *typexp*$_k$]

Filtrage et constructeurs fonctionnels :

```
# let get_values ca c =
  match c with
      Null -> invalid_arg "get_value"
    | Make(Valet, co) -> if (ca = co) then 20 else 2
    | Make(Neuf, co) -> if (ca = co) then 14 else 0
    | Make(As, _) -> 11
    | Make(Dix, _) -> 10
    | Make(Roi, _) -> 4
    | Make(Dame, _) -> 3
    | _ -> 0
;;
val get_values : color -> t -> int = <fun>
```

Remarquer : type inféré pour `ca`.

Variante pour les cartes : pas de type pour les figures, chaque figure est un constructeur fonctionnel du type des couleurs vers le type des cartes.

```
type t =
    As of color
  | Roi of color
  | Dame of color
  | Valet of color
  | Dix of color
  | Neuf of color
  | Huit of color
  | Sept of color

# let get_value ca c =
    match c with
        As(_) -> 11
      | Roi(_) -> 4
      | Dame(_) -> 3
      | Valet(co) -> if co = ca then 20 else 2
      | Dix(_) -> 10
      | Neuf(co) -> if co = ca then 14 else 0
      | _ -> 0
;;
val get_value : color -> t -> int = <fun>
```

Syntaxe: `match` *expr* `with` pat_1 `->` $expr_1$ `|` ... `|` pat_n `->` $expr_n$

Typage
– *expr*, pat_1, ..., pat_n sont du même type ;
– $expr_1$, ..., $expr_n$ sont du même type et c'est le type de toute l'expression

Évaluation, contrôle valeur de $expr_i$ si pat_i est le *premier motif* correspondant à la (forme de la) valeur de *expr*

Type avec paramètre `type 'a option = None | Some of 'a`
The type of optional values of type 'a.

Type récursif les S-expressions

```
type 'a s_exp =
    Nil
  | Atom of 'a s_exp
  | Cons of 'a s_exp * 'a s_exp
```

Structure arborescente (voir 8)

Exemples : concaténation et S-expressions "plates"

```
# let rec append e1 e2 =
  match e1 with
      Nil -> e2
    | Atom _ -> Cons(e1,e2)
    | Cons(e11,e12) -> Cons(e11, (append e12 e2)) ;;
val append : 'a s_exp -> 'a s_exp -> 'a s_exp = <fun>
# let rec flatten e =
  match e with
      Cons(e1, e2) -> (append (flatten e1) (flatten e2))
    | _ -> e ;;
val flatten : 'a s_exp -> 'a s_exp = <fun>
```

Noter : l'utilisation économique du filtrage, des noms.

6 Quelques structures standard

6.1 Les piles

Image : la pile d'assiettes, on rajoute une assiette au sommet, on retire une assiette du sommet ; *LIFO*, *L*ast *I*n *F*irst *O*ut.

Spécification abstraite Spécification fonctionnelle des opérations (indépendante d'une représentation), spécification *algébrique*. Le type des piles est *stack*, le type des éléments stockés est *elt*.

Symbols :
create : → stack
push : elt, stack → stack
top : stack → elt
pop : stack → stack
empty : stack → bool
Axioms : $\forall$ *s : stack ; x : elt*
(empty create) = true
(top (push x s)) = x
(pop (push x s)) = s

Interface du module `Stack` Fichier `stack.mli` de la distribution Objective Caml version 3.06 (bibliothèque standard) – extrait :

```
(** Last-in first-out stacks.

   This module implements stacks (LIFOs), with in-place modification.
*)

type 'a t
(** The type of stacks containing elements of type ['a]. *)

exception Empty
(** Raised when {!Stack.pop} or {!Stack.top} is applied to an empty stack. *)


val create : unit -> 'a t
(** Return a new stack, initially empty. *)

val push : 'a -> 'a t -> unit
(** [push x s] adds the element [x] at the top of stack [s]. *)

val pop : 'a t -> 'a
(** [pop s] removes and returns the topmost element in stack [s],
   or raises [Empty] if the stack is empty. *)

val top : 'a t -> 'a
(** [top s] returns the topmost element in stack [s],
   or raises [Empty] if the stack is empty. *)
```

```
val is_empty : 'a t -> bool
(** Return [true] if the given stack is empty, [false] otherwise. *)
```

S'éloigne un peu de la spécification fonctionnelle : "in-place modification". L'implentation utilise les listes (du module `list`) et les références.

Application calculette post-fixe

6.2 Files d'attente

Image : file d'attente, premier arrivé, premier servi ; *FIFO* *F*irst *I*n *F*irst *O*ut.

Spécification abstraite Le type des files d'attentes est *queue*, les opérations sont similaires à celle des piles : création/ajout/retrait, mais leur comportement diffère.

Symbols :
create : → queue
add : queue, elt → queue
front : queue → elt
rem : queue → queue
empty : queue → bool

Axioms : $\forall$ *s : queue ; x : elt*
(empty create) = true
(front (add create x)) = x
(front (add q x)) = (front q), si (empty q) = false
(rem (add create x)) = create
(rem (add q x)) = (add (rem q) x), si (empty q) = false

Noter les équations conditionnelles et récursives : une implémentation raisonnable *devra* s'en éloigner tout en respectant les *valeurs* calculées par les opérations.

Interface du module `Queue` Fichier `queue.mli` de la distribution Objective Caml version 3.06 (bibliothèque standard) – extrait :

```
(** First-in first-out queues.

   This module implements queues (FIFOs), with in-place modification.
*)

type 'a t
(** The type of queues containing elements of type ['a]. *)


exception Empty
(** Raised when {!Queue.take} or {!Queue.peek} is applied to an empty queue. *)


val create : unit -> 'a t
(** Return a new queue, initially empty. *)

val add : 'a -> 'a t -> unit
```

```
(** [add x q] adds the element [x] at the end of the queue [q]. *)

val take : 'a t -> 'a
(** [take q] removes and returns the first element in queue [q],
   or raises [Empty] if the queue is empty. *)

val peek : 'a t -> 'a
(** [peek q] returns the first element in queue [q], without removing
   it from the queue, or raises [Empty] if the queue is empty. *)

val is_empty : 'a t -> bool
(** Return [true] if the given queue is empty, [false] otherwise. *)
```

L'implantation, pour réaliser les opérations d'ajout et retrait en temps constant utilise une structure de *liste circulaire* (voir 7.3)

6.3 Structures associatives

Relation (éventuellement fonctionnelle) entre un ensemble de *clefs* et un ensemble de *valeurs*.

Spécification algébrique, type *map* avec *key* le type des clefs et *val* le type des valeurs

Symbols :

create : → map
add : map, key, val → map
rem : map, key → map
mem : map, key → bool
find : map, key → val

Axioms : $\forall$ *m : map ; k, k' : key ; v : val*

(rem create k) = create
(rem (add m k v) k) = m
(rem (add m k' v) k) = (add (rem m k) k' v), if $k \neq k'$
(rem (rem m k) k) = (rem m k)
(rem (rem m k') k) = (rem (rem m k) k' v), if $k \neq k'$
(mem create k) = false
(mem (rem m k v) k) = false
(mem (add m k v) k) = true
(mem (add m k' v) k) = (mem m k), if $k \neq k'$
(find (add m k v) k) = v
(find (add m k' v) k) = (find m k), if $k \neq k'$

Théorème : *create* et *add* sont *constructeurs* (ie. définissent l'ensemble des valeurs - termes canoniques).

Conséquence : l'equation *(mem (rem m k v) k) = false* n'est pas utile.

6.3.1 Listes d'association

Liste de couples (k, v) appelés *liaisons*, avec k : la clef et v : la valeur (associée à la clef).

Type concret : `('a * 'b) list`.

Opérations (module `List`)

```
val assoc : 'a -> ('a * 'b) list -> 'b
```

`(assoc a l)` *returns the value associated with key* `a` *in the list of pairs* `l`*. That is,* `(assoc a [ ... ; (a,b) ; ...] = b)` *if* `(a,b)` *is the leftmost binding of* `a` *in list* `l`*. Raise* `Not_found` *if there is no value associated with* `a` *in the list* `l`

`val mem_assoc : 'a -> ('a * 'b) list -> bool`
Same as `assoc`*, but simply return* `true` *if a binding exists, and* `false` *if no bindings exist for the given key.*

`val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`
`(remove_assoc a l)`
r
eturns the list of pairs `l` *without the first pair with key* `a`*, if any. Not tail-recursive.*

Les opérations d'initialisation et d'ajout sont celles des listes : constructeurs `[]` et `::`.

6.3.2 Tables de hachage

Principe : disposer sur l'ensemble des clefs (type `'a`) d'une *fonction de hachage* h de type `'a -> int` ; pour une liaison (k, v), la valeur v est stockée à l'indice $h(k)$ de la table (tableau). Temps d'accès constant.

Type abstrait : module `Hashtbl`

Opérations

`val create : int -> ('a,'b) t`
`(Hashtbl.create n)` *creates a new, empty hash table, with initial size* `n`*. For best results,* `n` *should be on the order of the expected number of elements that will be in the table. The table grows as needed, so* `n` *is just an initial guess.*

`val add : ('a, 'b) t -> 'a -> 'b -> unit`
`(Hashtbl.add tbl x y)` *adds a binding of* `x` *to* `y` *in table* `tbl`*. Previous bindings for* `x` *are not removed, but simply hidden. That is, after performing* `(Hashtbl.remove tbl x)`*, the previous binding for* `x`*, if any, is restored. (Same behavior as with association lists.)*

`val find : ('a, 'b) t -> 'a -> 'b`
`(Hashtbl.find tbl x)` *returns the current binding of* `x` *in* `tbl`*, or raises* `Not_found` *if no such binding exists.*

`val mem : ('a, 'b) t -> 'a -> bool`
`(Hashtbl.mem tbl x)` *checks if* `x` *is bound in* `tbl`*.*

Application ???

<<<

6.4 Les listes à "curseur" (*zip-listes*)

```
(** Zip lists

  This module implements "zip-list" with  access at a "current location"
  inside the list.
*)

type 'a t
(** type of zip-list *)

exception Empty
```

```
(** Raised when unxecpected empty list is given as argument *)

val create : unit -> 'a t
(** Return a new empty zip-list *)

val is_empty : 'a t -> bool
(** Return [true] is the given zip-list is empty, [false] otherwise *)

val left_end : 'a t -> bool
(** Return [true] is the current location is at the beginning of the
  given zip-list *)


val rigth_end : 'a t -> bool
(** Return [true] is the current location is at the end of the
  given zip-list *)

val get_elt : 'a t -> 'a
(** Return the element at the current location *)

val step_forward : 'a t -> 'a t
(** Return a zip-list whose current location is one step on right
  of the one of the given zip-list. Raise [Empty] if the given zip-list
  is empty or at its current location is at its end. *)

val step_backward : 'a t -> 'a t
(** Return a zip-list whose current location is one step on left
  of the one of the given zip-list. Raise [Empty] if the given zip-list
  is empty or at its current location is at its beginning. *)

val ins_after : 'a -> 'a t -> 'a t
(** [ins_after x xs] return a new zip-list where [x] is added to [xs] at the
  right of the current location of [xs]. The current location of the result is
  on the added element. *)

val ins_before : 'a -> 'a t -> 'a t
(** [ins_after x xs] return a new zip-list where [x] is added to [xs] at the
  left of the current location of [xs]. The current location of the result is
  on the added element. *)

val find_forward : ('a t -> bool) -> 'a t -> 'a t
(** [find-forward f xs] return the first sub list found at the right
  of the current location which satisfies [f]. Return a zip-list with
  current location at its end if no such element exists. Return [xs]
  if [xs] is empty. *)

val find_backward : ('a t -> bool) -> 'a t -> 'a t
(** [find-forward f xs] return the first sub list found at the left
  of the current location which satisfies [f]. Return a zip-list with
  current location at its beginning if no such element exists. Return [xs]
  if [xs] is empty. *)
```

```
val to_list : 'a t -> 'a list
(** Return the list of the element of the given zip-list *)
```

Type concret :

```
type 'a t =
  {
    backward: 'a list;
    forward: 'a list
  }
```

Exos : le même avec modifications en place

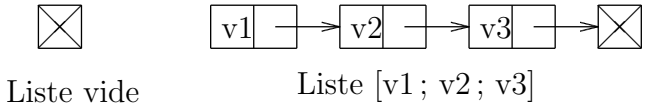
>>>

7 Variations sur les listes chaînées

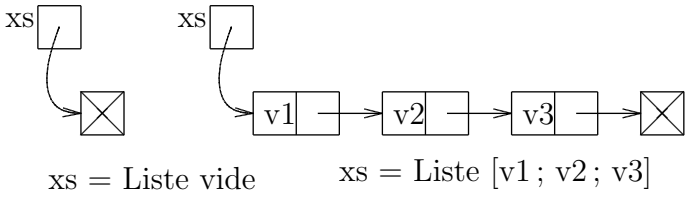
Un manière d'implanter les listes à l'aide d'enregistrements et de références.

7.1 Listes simplement chaînées

Une liste est :
- soit une liste vide
- soit une *cellule* contenant le premier élément de la liste et un *pointeur* vers (une référence à) la suite de la liste.



Uniformité : une liste est toujours une référence :



Type concret Pour implanter *soit vide, soit cellule...* on utilise (type prédéfini)

`type 'a option = None | Some of 'a`
The type of optional values of type 'a.

où `None` vaudra pour *vide*.

Définition en deux temps : cellule (enregistrement), puis référence à une cellule (type principal)

```
type 'a cell =
  {
    elt : 'a;
    next : 'a cell option ref
  }

type 'a t = 'a cell option ref
```

Noter : le type de `next` est ágal à `'a t`

Alternative : définitions *mutuellement récursive*

```
type 'a cell =
  {
    elt : 'a;
    next : 'a t
  }

and 'a t = 'a cell option ref
```

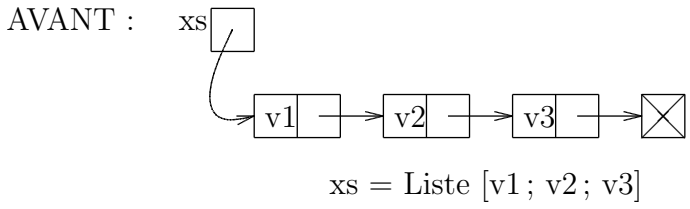
Constructeurs pour la liste vide, on prévoit aussi une exception et une fonction de test.

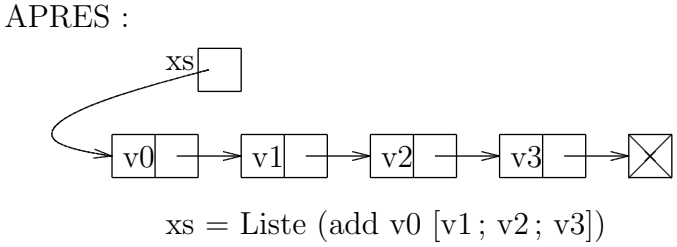
Liste vide

```
exception Empty

let create () = ref None

let is_empty xs = (!xs = None)
```

Ajout en place : modifie la valeur référencée, mais pas la référence elle-même.

```
val add : 'a -> 'a t -> unit
(** [add x xs] sets [xs] to the list whose head is [x] and tail
    the given [xs].                                          *)
```






```
let add x xs =
  xs := Some { elt = x; next = ref !xs }
```

Accesseurs on leur donne les signatures

```
val get_elt : 'a t -> 'a
(** returns the value at the head of the list. Raise [Empty]
    if its argument is an empty clist                        *)

val get_next : 'a t -> 'a t
(** returns the tail of the list. Raise [Empty] if its argument
    is an empty clist                                        *)
```

L'implantation est immédiate

```
let get_elt xs  =
  match !xs with
      None -> raise Empty
    | Some c -> c.elt

let get_next xs  =
  match !xs with
      None -> raise Empty
    | Some c -> c.next
```

Un itérateur sur le modèle de l'`iter` du module `List`

```
val iter : ('a t -> unit) -> 'a t -> unit
(** [iter f xs] applies [f] in turn to elements of [xs].          *)
```

On en a facilement une implantation récursive terminale

```
let rec iter f xs =
  if (is_empty xs) then ()
  else ( (f xs) ; (iter f (get_next xs)) )
```

On peut l'utiliser pour afficher les éléments d'une liste (ici, d'entiers)

```
# let xs = create() ;;
val xs : '_a option ref = {contents = None}
# add 8 xs; add 6 xs; add 4 xs; add 2 xs;;
- : unit = ()
# iter (fun ys -> Printf.printf"(%d)" (get_elt ys)) xs ;;
(2)(4)(6)(8)- : unit = ()
```

Autre utilisation (amusante) de la fonctionnelle `iter` : la fonction `length`

```
let length xs =
  let n = ref 0 in
    iter (fun _ -> incr n) xs;
    !n
```

Un autre itérateur Spécification

```
val find : ('a t -> bool) -> 'a t -> 'a t
(** [find c xs] returns the leftmost sublist of [xs] that
    verifies [c] or an empty list if no such sublist exists. *)
```

Implantation récursive (terminale)

```
let rec find c xs =
  if (is_empty xs) or (c xs) then
    xs
  else
    (find c (get_next xs))
```

On peut aussi en donner une version itérative, mais attention au jeu des références couplé avec les modifications. Une version correcte est

```
let find c xs =
  let xs' = ref xs in
    while not (is_empty !xs') && not (c (get_elt !xs')) do
      xs' := get_next !xs'
    done;
    !xs'
```

Utilise une référence auxiliaire locale pour "parcourir" la liste. Sinon

```
let find c xs =
  while not (is_empty !xs) && not (c (get_elt !xs)) do
    xs := get_next !xs
  done;
  !xs
```

l'argument peut-être modifié (effet de bord).

Noter : ça n'a pas lieu avec la version récursive ; *passage des arguments par valeur.*

Remarque : écrire

```
let find c xs =
  let xs' = xs in
    while not (is_empty xs') && not (c (get_elt xs')) do
      xs' := get_next xs'
    done;
    !xs'
```

aurait le même effet de bord désastreux.

Exemple d'utilisation : la fonction `mem` *à la Scheme*.

```
val mem : 'a -> 'a t -> 'a t
(** [mem x xs] returns the lestmost sublist of [xs] whose head is
    equal to [x] or an empty list if no such sublist exists.      *)
```

Implantation

```
let mem x xs =
  find (fun ys -> (get_elt ys) = x) xs
```

Fonctions d'insertion on veut définir des fonctions d'insertion, pas nécessairement en tête, mais à une position donnée par un certain critère.

```
val ins_before : 'a -> ('a t -> bool) -> 'a t -> unit
(** [ins_before x c xs] set [xs] to the list obtained by inserting
    [x] before the head of the leftmost sublist which verifies [c]
    or at the end of the given [xs] if no such sublist exists.    *)

val ins_after : 'a -> ('a t -> bool) -> 'a t -> unit
(** [ins_after x c xs] set [xs] to the list obtained by inserting
    [x] after the head of the leftmost sublist which verifies [c]
    or at the end of the given [xs] if no such sublist exists.    *)
```

Implantation : trouver, grace à `find`, l'endroit où insérer ; prendre garde, si besoin, à la liste vide.

```
let ins_before x c xs =
  let xs' = find c xs in
    add x xs'

let ins_after x c xs =
  let xs' = find c xs in
    if (is_empty xs') then
      add x xs'
    else
      add x (get_next xs')
```

Exemple : insèrer 5 entre 4 et 6, dans la liste (ordonnée) qui contient 2, 4, 6, 8 ; c'est à dire *avant* 6. Exprimer le bon critère : *la sous-liste dont l'élément de tête est supérieur à 5.*

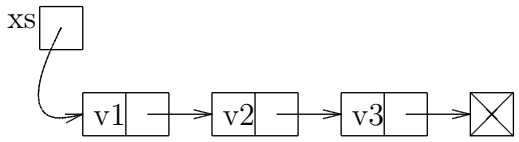
```
# ins_before 5 (fun ys -> (get_elt ys) > 5) xs ;;
- : unit = ()
# iter (fun ys -> Printf.printf"(%d)" (get_elt ys)) xs ;;
(2)(4)(5)(6)(8)- : unit = ()
```

Fin de liste ajouter un élément en fin de liste

```
val add_last : 'a -> 'a t -> unit
(** [add x xs] set [xs] to the list obtained by adding [x] as
```

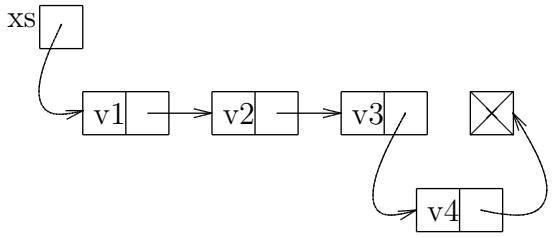
```
the last element of the given [xs].                    *)
```

AVANT :



xs = Liste [v1 ; v2 ; v3]

APRES :



xs = Liste (add_last [v1 ; v2 ; v3] v4)

Trouver la (sous) liste vide finissant une liste

```
val get_last : 'a t -> 'a t
(** returns the endind empty sublist of its argument.      *)
```

Implantation avec `find`

```
let get_last xs =
   find (fun _ -> false) xs
```

Modifier la liste vide finale

```
let add_last xs x =
  (add x (get_last xs))
```

Exemple :

```
# let xs = create() ;;
val xs : '_a option ref = {contents = None}
# add_last xs 1; add_last xs 2 ;;
- : unit = ()
# iter (fun ys -> Printf.printf"(%d)" (get_elt ys)) xs ;;
(1)(2)- : unit = ()
```

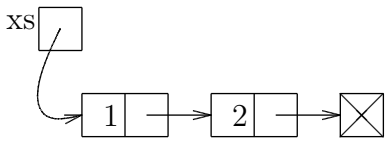
On peut aussi ajouter une autre liste en fin de liste

```
let append xs ys =
  (get_last xs) := !ys
```

Mais attention alors aux effets de bord :

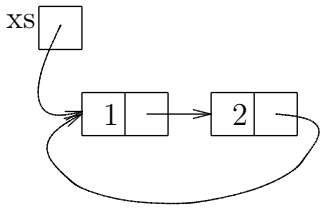
```
# append xs xs ;;
- : unit = ()
# iter (fun ys -> Printf.printf"(%d)" (get_elt ys)) xs ;;
(1)(2)(1)(2)(1)(2)(1)(2)(1)(2)(1)(2)(1)(Interrupted.
```

AVANT :



xs = Liste [1 ; 2]

APRES :



xs = Liste (append xs xs)

Copie créer une nouvelle structure chaînée contenant les *mêmes* éléments.

```
val copy : 'a t -> 'a t
(** returns a fresh copy of its argument                    *)
```

Implantation récursive, facile, mais non terminale

```
let rec copy xs =
  if (is_empty xs) then
    create()
  else
    let xs' = copy (get_next xs) in
      add (get_elt xs) xs';
      xs'
```

Implantation récursive : trois *pointeurs* ; un pour le résultat, un pour l'ajout (en fin), un pour le parcours.

```
let copy xs =
  let first = ref (create()) in
  let last = ref !first in
  let xs' = ref xs in
    while not (is_empty !xs') do
      add (get_elt !xs') !last;
      last := (get_next !last);
      xs' := (get_next !xs')
    done;
    !first
```

Exemple :

```
# let xs = create() ;;
val xs : '_a option ref = {contents = None}
# add_last xs 1; add_last xs 2 ;;
- : unit = ()
# append xs (copy xs) ;;
- : unit = ()
# iter (fun ys -> Printf.printf"(%d)" (get_elt ys)) xs ;;
```

```
(1)(2)(1)(2)- : unit = ()
```

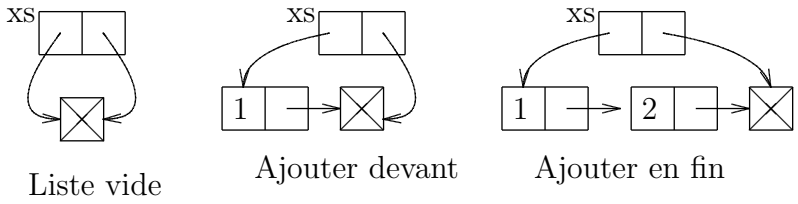
7.2 La chandelle par les deux bouts

Une structure qui soit pile et file en utilisant `Clist`.

Type concret On maintient une référence sur le dernier (champ modifiable)

```
type 'a t =
    {
      first : 'a Clist.t;
      mutable last : 'a Clist.t
    }
```

Constructeurs illustration graphique



Liste vide : partagent la même valeur

```
let create () =
  let xs = Clist.create() in
    { first = xs; last = xs }
```

Ajout en tête et en fin : cas particulier pour l'ajout en tête la première fois (liste vide)

```
let add_first x xs =
  Clist.add x xs.first;
  if not (Clist.is_empty xs.last) then
    xs.last <- (Clist.get_next xs.last)

let add_last xs x =
  Clist.add x xs.last;
  xs.last <- (Clist.get_next xs.last)
```

Petit auxiliaire d'affichage (liste d'entiers)

```
let print_ints xs =
  let f n = Printf.printf"(%d)" n in
    Clist.iter (fun xs -> f (Clist.get_elt xs)) xs.first;
    print_newline()
```

Exemple :

```
# let xs = create() ;;
val xs : '_a t = {first = <abstr>; last = <abstr>}
# add_first 0 xs; add_last xs 1; add_first (-1) xs ;;
- : unit = ()
# print_ints xs ;;
(-1)(0)(1)
- : unit = ()
```

Remarque, on peut aussi commencer par `add_last`

```
# let xs = create() ;;
val xs : '_a t = {first = <abstr>; last = <abstr>}
# add_last xs 0; add_first 1 xs; add_last xs (-1) ;;
- : unit = ()
# print_ints xs ;;
(-1)(0)(1)
- : unit = ()
```

Destructeurs on retrouve l'opération `pop` des piles ou `take` des files d'attente

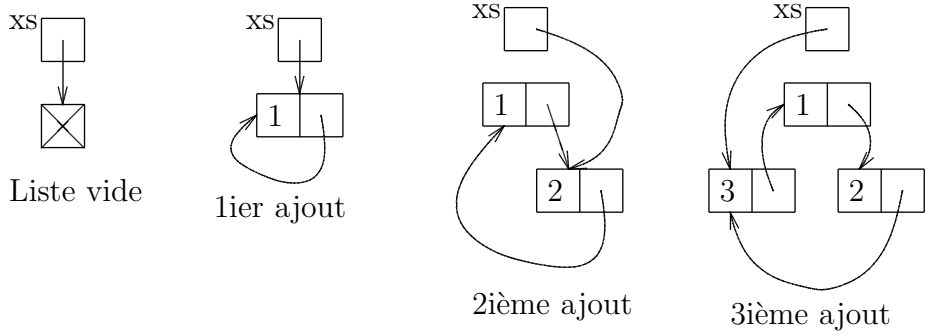
```
let rem_first xs =
  if (is_empty xs) then
    raise Empty
  else
    let x = (Clist.get_elt xs.first) in
      xs.first <- (Clist.get_next xs.first);
      x
```

Noter : on se souvient du premier élément *avant* de retirer la première cellule.

En revanche, on n'a pas d'accès *directe* au dernier èlément : `last` nous donne l'ultime sous-liste toujours vide et on ne sait pas remonter à la cellule précédente.

7.3 Listes circulaires

Idée : on maintient une référence sur la dernière cellule ajoutée (*last*) et la dernière cellule ajoutée pointe sur la première.



Type concret plus besoin d'un champs `next` de type `option`

```
type 'a cell =
    {
      elt : 'a;
      mutable next : 'a cell
    }

type 'a t = 'a cell option ref
```

Constructeurs la file vide (avec son exception, test)

```
exception Empty

let create () = ref None

let is_empty xs =
  match !xs with
      None -> true
    | _ -> false
```

L'opération d'ajout : cas particulier pour premier ajout (liste vide)

```
let add x xs =
  match !xs with
      None -> (
let rec c = { elt = x; next = c } in
  xs := Some c
      )
    | Some clast -> (
let cnew = { elt = x; next = clast.next } in
  clast.next <- cnew;
  xs := Some cnew
      )
```

Parcours utilisation de *l'égalité physique*. Fonction qui transforme une file en liste (type `'a list`.

```
let to_list xs =
  match !xs with
      None -> []
    | Some clast -> (
let rec loop c =
  if c == clast then
    [c.elt]
  else
    c.elt::(loop c.next)
in
  loop clast.next
      )
```

Exemple :

```
# let xs = create();;
val xs : '_a option ref = {contents = None}
# to_list xs;;
- : '_a list = []
# add 1 xs;;
- : unit = ()
# to_list xs;;
- : int list = [1]
# add 2 xs;;
- : unit = ()
# to_list xs;;
- : int list = [1; 2]
# add 3 xs;;
- : unit = ()
# to_list xs;;
```

```
- : int list = [1; 2; 3]
```

Opération de retrait le `take` du module `Queue`. Cas particulier pour les file à un seul élément (utilisation de l'égalité physique).

```
let take xs =
  match !xs with
      None -> raise Empty
    | Some clast -> (
let cfirst = clast.next in
  (if clast == cfirst then xs := None
   else clast.next <- cfirst.next);
  cfirst.elt
      )
```

Exemple et test :

```
# let xs = create();;
val xs : '_a option ref = {contents = None}
# add 1 xs; add 2 xs; add 3 xs;;
- : unit = ()
# to_list xs;;
- : int list = [1; 2; 3]
# take xs;;
- : int = 1
# to_list xs;;
- : int list = [2; 3]
# take xs;;
- : int = 2
# to_list xs;;
- : int list = [3]
# take xs;;
- : int = 3
# to_list xs;;
- : int list = []
# take xs;;
Exception: Empty.
```

7.4 Listes doublement chaînées

Structure inspirée des listes simplement chaînées, mais avec deux références : précesseur et successeur.

Type concret

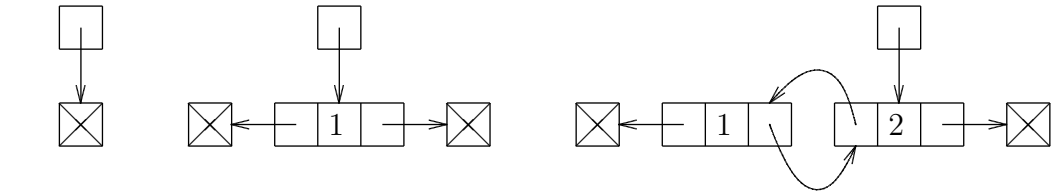
```
type 'a cell =
  {
    elt : 'a;
    next : 'a cell option ref;
    prev : 'a cell option ref
  }

type 'a t = 'a cell option ref
```

Liste vide et accesseurs

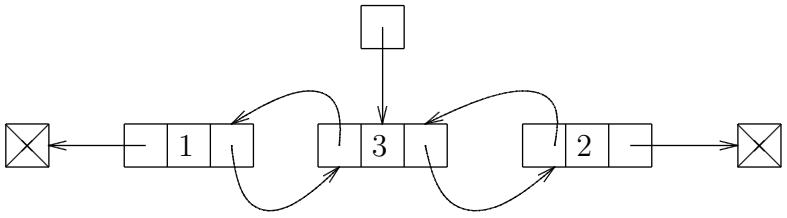
```
exception Empty

let create () = ref None

let is_empty xs = (!xs = None)

let get_elt xs =
  match !xs with
    None -> raise Empty
  | Some c -> c.elt

let get_next xs =
  match !xs with
    None -> raise Empty
  | Some c -> c.next

let get_prev xs =
  match !xs with
    None -> raise Empty
  | Some c -> c.prev
```

Opération d'insertion modification *en place*

Illustration graphique



Liste vide 1ier ajout 2ieme ajout (apres)



3ieme ajout (avant)

On tient la liste par le "milieu" : dernier élélement inséré, *lément courant* ou *curseur*.

Deux opérations selon que l'insère avant ou aprés l'élélement courant.

```
let ins_after x xs =
  match !xs with
    None -> (
```

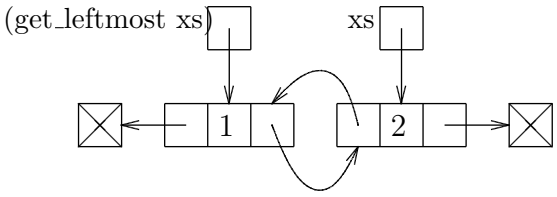
```
      xs := Some { elt = x; next = ref None; prev = ref None }
    )
  | Some c -> (
      let cnew = { elt = x; prev = ref !xs; next = ref !(c.next) } in
      let s_cnew = Some cnew in
        c.next := s_cnew;
        (match !(cnew.next) with
           Some cnext -> cnext.prev := s_cnew
         | _ -> ());
        xs := s_cnew
    )

let ins_before x xs =
  match !xs with
    None -> (
      xs := Some { elt = x; next = ref None; prev = ref None }
    )
  | Some c -> (
      let cnew = { elt = x; prev = ref !(c.prev); next = ref !xs } in
      let s_cnew = Some cnew in
        c.prev := s_cnew;
        (match !(cnew.prev) with
           Some cprev -> cprev.next := s_cnew
         | _ -> ());
        xs := s_cnew
    )
```

Tenir compte des cas particuliers : liste vide, insertion avant l'élément le plus à gauche, après l'élément le plus à droite.

Autre accesseur et utilitaire de conversion Accès au "premier élément", l'élément le plus à gauche

```
let get_leftmost xs =
  let rec loop xs =
    let prev = (get_prev xs) in
      if (is_empty prev) then
        xs
      else
        (loop prev)
  in
    if (is_empty xs) then
      xs
    else
      (loop xs)
```

Attention au partage



Conversion en liste Objective Caml

```
let to_list xs =
  let rec loop xs =
    if (is_empty xs) then []
    else (get_elt xs)::(loop (get_next xs))
  in
    loop (get_leftmost xs)
```

Exemple et test

```
# let xs = create();;
val xs : '_a option ref = {contents = None}
# for i=1 to 5 do (ins_after i xs) done ;;
- : unit = ()
# (to_list xs) ;;
- : int list = [1; 2; 3; 4; 5]
```

Opération de recherche déplace le curseur á gauche ou à droite, modification de l'élément courant (procédure et non fonction), critère d'arrêt passé en argument.

```
let move_forward f xs =
  if not (is_empty xs) then
    while not ((is_empty (get_next xs)) or (f xs)) do
      xs := !(get_next xs)
    done

let move_backward f xs =
  if not (is_empty xs) then
    while not ((is_empty (get_prev xs)) or (f xs)) do
      xs := !(get_prev xs)
    done
```

Exemple d'utilisation : revenir en début de liste (élément le plus à gauche)

```
let rewind xs =
  search_back (fun _ -> false) xs
```

Test

```
# let xs = create() ;;
val xs : '_a option ref = {contents = None}
# for i=1 to 5 do (ins_after i xs) done ;;
- : unit = ()
# get_elt xs ;;
- : int = 5
# rewind xs ;;
- : unit = ()
# get_elt xs ;;
- : int = 1
```

Application construire une liste ordonnée à partir d'élements lus sur l'entrée standard

```
let lt x xs = (get_elt xs) < x
let gt x xs = (get_elt xs) > x
let rec loop xs =
```

```
  print_string "? "; flush stdout;
  let x = read_int() in
    if (is_empty xs) then (
      (ins_after x xs);
      (loop xs)
    ) else if (x < (get_elt xs)) then (
        (move_backward (lt x) xs);
        (if ((get_elt xs) < x) then (ins_after x xs) else (ins_before x xs));
        (loop xs)
    ) else (
        (move_forward (gt x) xs);
        (if ((get_elt xs) < x) then (ins_after x xs) else (ins_before x xs));
        (loop xs)
    )
;;
```

Toujours les cas particuliers : liste vide, élément le plus à gauche, le plus à droite.

Exemple et test, on sort sur *fin de fichier* (ctrl-D)

```
# let xs = create ();;
val xs : '_a option ref = {contents = None}
# loop xs ;;
? 6
? 3
? 9
? 7
? 2
? 8
? 5
? 1
? 4
? Exception: End_of_file.
# to_list xs;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Représentation alternative

```
type 'a t =
  Empty
| Left_most of 'a t
| Right_most of 'a t
| Medium of 'a t * 'a * 'a t
```

8 Les arbres

Ensemble de *nœuds*, dont une *racine*. Relation de *parenté* entre nœuds (*ascendance*, *descendance*), structure *hierarchique*. Nœuds sans descendants : *feuilles*. L'ensemble peut être vide, arbre vide.

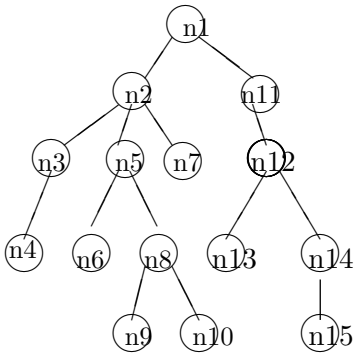
Donnée associée au nœud : *étiquettes*. Tous les nœuds ou uniquement aux feuilles.

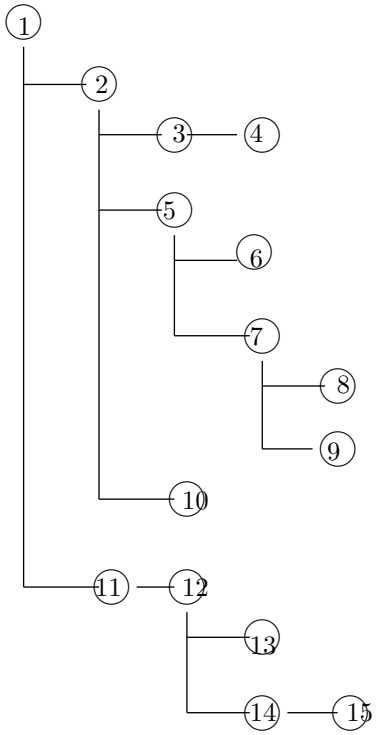
Chemin : suite de nœuds $n1, \ldots, n_k$, avec n_i ascendant de n_{i+1}.

Branche : chemin $n_0, n1, \ldots, n_k$, avec n_0 racine, n_k feuille.

Profondeur d'un nœud n, longueur du chemin $n_0, \ldots, n$ avec n_0 racine.

Représentations graphiques des arbres, cercles : nœuds, traits : relation de parenté.





Vision algébrique définition *récursive* (tous nœuds étiquetés).

Type : $tree[A]$, avec A type de étiquettes
– l'arbre vide $Empty : \rightarrow tree$;
– une famille de constructeurs de *branchement* $Br_0, Br_1, \ldots$ avec pour chaque $i \in \mathbb{N}, Br_i : A \times (tree[A])^i \rightarrow tree[A]$, avec $(tree[A])^i$, i-uplet d'arbres.
Exemple (écriture - presque - complètement *indentée*)

```
Br2(n1,
    Br3(n2,
        Br1(n3, Br0(n4)),
        Br2(n5,
            Br0(n6)),
```

Br_2(n7,
Br_0(n8),
Br_0(n9)),
Br_0(n10)),
Br_1(n11,
Br_2(n12,
Br_0(n13),
Br_1(n14, Br_0(n15)))))

Dans $Br_i(n, t_1, \ldots, t_i)$, les t_i sont appellés *sous-arbres (immédiats)*.

Attention aux cas limites : la feuille $Br_0(n)$ peut être représentée par $Br_k(N, Empty, \ldots, Empty)$. Feuille : pas de sous-arbres *ou* tous sous-arbres vides.

Nota : pour obtenir étiquettes aux feuilles uniquement : $Br_0 : \ A \to tree[A]$ et $Br_i : \ (tree[A])^i \to tree[A]$ pour $i > 0$.

8.1 Arbres binaires

Un seul constructeur de branchement : Br_2.

Type concret :

```
type 'a btree =
  Nil
| Br of 'a * 'a btree 'a btree
```

Arbre vide :

```
exception Empty
```

Feuille : `Br(a, Nil, Nil)`

```
let is_leaf t =
  match t with
    Br(_, Nil, Nil) -> true
  | _ -> false
```

Définition récursive : fonctions naturellement récursives.

Calcul du nombre de nœuds (taille) :

```
let rec size t =
  match t with
    Nil -> 0
  | Br(_, t1, t2) -> (size t1) + (size t2)
```

Hauteur d'un arbre (taille de la plus longue branche).

```
let rec height t =
  match t with
    Nil -> 0
  | Br(_, t1, t2) -> (succ (max (height t1) (height t2)))
```

Itérateur :

```
# let rec fold_right f t b =
    match t with
      Nil -> b
    | Br(a0, t1, t2) -> (f a0 (fold_right f t1 b) (fold_right f t2 b))
  ;;
val fold_right : ('a -> 'b -> 'b -> 'b) -> 'a btree -> 'b -> 'b = <fun>
```

Énumération des nœuds : en *profondeur*, de gauche à droite, de droite à gauche ou "par le milieu" (postfixe, préfixe, infixe). Exemples :

```
let rec to_list_prefix t =
  match t with
    Nil -> []
  | Br(a, t1, t2) -> a::((to_list_prefix t1) @ (to_list_prefix t2))

let rec to_list_postfix t =
  match t with
    Nil -> []
  | Br(a, t1, t2) -> ((to_list_postfix t1) @ (to_list_postfix t2) @ [a])

let rec to_list_infix t =
  match t with
    Nil -> []
  | Br(a, t1, t2) -> ((to_list_infix t1) @ [a] @ (to_list_infix t2))
```

Avec l'itérateur

L'astuce du peigne : modifier l'arbre (pour préfixe, remonter le sous-arbre gauche), un seul appel récursif.

```
let rec to_list_prefix t =
  match t with
    Nil -> []
  | Br(a, Nil, t2) -> a::(to_list_prefix t1)
  | Br(a1, Br(a2, t1, t2), t3) -> (to_list_prefix (Br(a1, t1, Br(a2, t2, t3))))
```

Ènumération *en largeur* : utilisation d'une file d'attente

```
let to_list_by_width t =
  let rec loop q =
    try
      match (Queue.take q) with
        Nil -> (loop q)
      | Br(a, t1, t2) -> (
          (Queue.add t1 q);
          (Queue.add t2 q);
          a::(loop q)
        )
    with
      Queue.Empty -> []
  in
    let q = Queue.create() in
      Queue.add t q;
      (loop q)
```

Recherche d'un élément : utilisation des exceptions.

Exemple : recherche d'un sous-arbre de racine donnée

```
# let rec search x t =
    match t with
      Nil -> raise Not_found
    | Br(a, t1, t2) ->
        if (x = a) then
          t
```

```
          else (
            try (search x t1)
            with Not_found -> (search x t2)
          );;
val search : 'a -> 'a btree -> 'a btree = <fun>
```

Exo : calcul de la profondeur d'un nœud.

Application : expressions arithmétiques

Un type pour les nœuds

```
type ar_node =
  Add
| Mul
| Cste of int
| Var of string
```

Quelques fonctions utilitaires

```
# let c i = Br(Cste i, Nil, Nil)
  let add t1 t2 = Br(Add,t1,t2)
  let mul t1 t2 = Br(Mul,t1,t2)
  ;;
val c : int -> ar_node btree = <fun>
val add : ar_node btree -> ar_node btree -> ar_node btree = <fun>
val mul : ar_node btree -> ar_node btree -> ar_node btree = <fun>
```

Codage de l'expression : $(((1 + 2) \times 3) + (4 \times 5)) + (6 \times 7)$

```
# let e =
    add (add (mul (add (c 1) (c 2)) (c 3)) (mul (c 4) (c 5))) (mul (c 6) (c 7))
  ;;
val e : ar_node btree =
  Br (Add,
   Br (Add,
    Br (Mul, Br (Add, Br (Cste 1, Nil, Nil), Br (Cste 2, Nil, Nil)),
     Br (Cste 3, Nil, Nil)),
    Br (Mul, Br (Cste 4, Nil, Nil), Br (Cste 5, Nil, Nil))),
   Br (Mul, Br (Cste 6, Nil, Nil), Br (Cste 7, Nil, Nil)))
```

Affichages :

```
# let print_ar_node n =
    match n with
      Add -> print_string " +"
    | Mul -> print_string " *"
    | Cste i -> Printf.printf " %d" i
    | Var x -> Printf.printf " %s" x
  ;;
val print_ar_node : ar_node -> unit = <fun>
# List.iter print_ar_node (to_list_postfix e); print_newline() ;;
 1 2 + 3 * 4 5 * + 6 7 * +
- : unit = ()
# List.iter print_ar_node (to_list_prefix e); print_newline() ;;
 + + * + 1 2 3 * 4 5 * 6 7
- : unit = ()
# List.iter print_ar_node (to_list_infix e); print_newline() ;;
```

```
 1 + 2 * 3 + 4 * 5 + 6 * 7
- : unit = ()
```

Noter : pour bien faire infixe, parenthèses (exo)

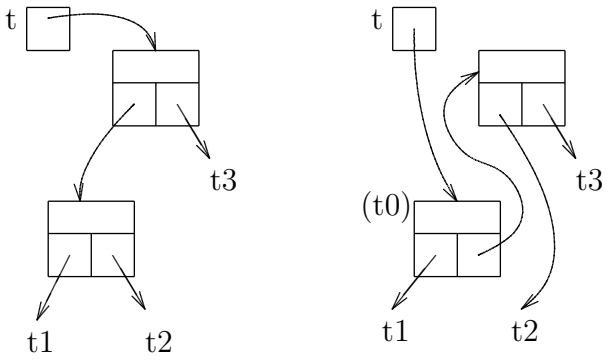
Arbres *"procéduraux"* Enregistrement et référence

```
type 'a cell =
    {
      elt : 'a;
      left : 'a cell option ref;
      right : 'a cell option ref
    }

type 'a t = 'a cell option ref

exception Empty

let create () = ref None

let is_empty t = (!xs = None)

let mk_node x t1 t2 =
  { elt = x; left = t1; right = t2 }

let get_elt t =
  if (is_empty t) then
    raise Empty
  else
    t.elt

let get_left t
  if (is_empty t) then
    raise Empty
  else
    t.left

let get_right t
  if (is_empty t) then
    raise Empty
  else
    t.right
```

Parcours infix

```
let rec to_list_infix t =
  if (is_empty t) then
    []
  else
    (to_list_infix (get_left t))@(get_elt t)::(to_list_infix (get_right t))
```

Restructuration en place (peigne)




```
let rec peigne t =
  if (is_empty t) then
     ()
  else
    let t0 = (get_left t) in
      if (is_empty t0) then
        (peigne (get_right t))
      else (
        let t0_c = !t0 in
        let t2 = (get_right t0) in
        let t2_c = !t2 in
          t2 := !t;
          t0 := t2_c;
          t := t0_c;
          (peigne t)
      )
```

Arbres binaires de recherche une relation (d'ordre) R sur les nœuds.

Définition récursive :

1. Nil est un arbre binaire de recherche.
2. $Br(a_0, t_1, t_2)$ est un arbre binaire de recherche ssi pour t_1 et t_2 sont des arbres binaires de recherche et si $t_1 = Br(a_1, t_{11}, t_{12})$ alors $a_1 R a_0$ et si $t_2 = Br(a_2, t_{21}, t_{22})$ alors $a_0 R a_2$.

Création d'un arbre binaire de recherhe : fonction d'insertion *ins* vérifiant l'*invariant* : *si t est un arbre binaire de recherche alors pour tout $n \in A$, $ins(a, t)$ est un arbre biniare de recherche.*

```
# let rec ins_bst r a0 t =
    match t with
      Nil -> Br(a0, Nil, Nil)
    | Br(a, t1, t2) ->
       if (r a0 a) then
         Br(a, (ins_bst r a0 t1), t2)
       else (* hyp: (r a a0) *)
         Br(a, t1, (ins_bst r a0 t2))
    ;;
val ins_bst : ('a -> 'a -> bool) -> 'a -> 'a btree -> 'a btree = <fun>
```

L'arbre croît par les feuilles.

Tests : utilisation de `Random.int`

```
# let random_int_list len m =
```

```
    let rec loop i ns =
      if i=len then
        ns
      else
        (loop (succ i) ((Random.int m)::ns))
    in
      (loop 1 [])
      ;;
val random_int_list : int -> int -> int list = <fun>
# let ns = (random_int_list 10 10) ;;
val ns : int list = [6; 3; 5; 1; 8; 1; 7; 1; 0]
# let t = (List.fold_right (ins_bst (<)) ns Nil) ;;
val t : int btree =
  Br (0, Nil,
   Br (1, Nil,
    Br (7,
     Br (1, Nil, Br (1, Nil, Br (5, Br (3, Nil, Nil), Br (6, Nil, Nil)))),
     Br (8, Nil, Nil))))
# to_list_infix t;;
- : int list = [0; 1; 1; 1; 3; 5; 6; 7; 8]
```

On a une fonction de tri sur les listes !

Recherche d'un élément : on sait *a priori* où doit être l'élément cherché ; racine, à gauche *ou* à droite. Un seul appel récursif et terminal.

```
# let rec bst_search r a t =
    match t with
      Nil -> raise Not_found
    | Br(a0, _, _) when (a = a0) -> t
    | Br(a0, t1, _) when (r a0 a) -> (bst_search r a t1)
    | Br(a0, _, t2) -> (bst_search r a t2)
  ;;
val bst_search : ('a -> 'a -> bool) -> 'a -> 'a btree -> 'a btree = <fun>
```

Noter : utilisation filtres gardés.

Suppression de la racine : faire remonter le plus grand des plus petits (ou le plus petit des plus grands)

Auxiliaires : donner l'élément maximal et supprimer l'élément maximal.

Naif : deux fonctions

```
let rec get_max t =
  match t with
    Nil -> raise Empty
  | Br(a, Nil, Nil) -> a
  | Br(a, t1, t2) -> (get_max t2)

let rec rem_max t =
  match t with
    Nil -> raise Empty
  | Br(a, Nil, Nil) -> Nil
  | Br(a, t1, t2) -> Br(a, t1, (rem_max t2))
```

Plus malin : une fonction qui calcule deux valeurs (en fait : *une valeur double*)

```
# let rec get'rem_max t =
    match t with
```

```
      Nil -> raise Empty
    | Br(a, Nil, Nil) -> (a, Nil)
    | Br(a, t1, t2) ->
       let (a', t2') = (get'rem_max t2) in
         (a', Br(a, t1, t2'))
  ;;
val get'rem_max : 'a btree -> 'a * 'a btree = <fun>
```

Supprimer la racine d'un arbre binaire de recherche : prendre garde aux cas limites (sous-arbres vides)

```
let bst_rem_root t =
  match t with
    Br(_, Nil, Nil) -> Nil
  | Br(_, Nil, t2) ->
      let (a, t2') = (get'rem_max t2) in
        Br(a, t2, Nil)
  | Br(_, t1, t2) ->
      let (a, t1') = (get'rem_max t1) in
        Br(a, t1', t2)
  | _ -> t
```

Suppression d'un élément : exo.

Tas arbre *équilibré* : la différence de profondeur entre deux feuilles est au maximum de 1

Soit R relation (d'ordre) sur les nœuds, un arbre t est un tas ssi

1. t est équilibré et
2. pour tout sous arbre t' de racine a_0, pour tout nœud $a \neq a_0$ de t', aRa_0.

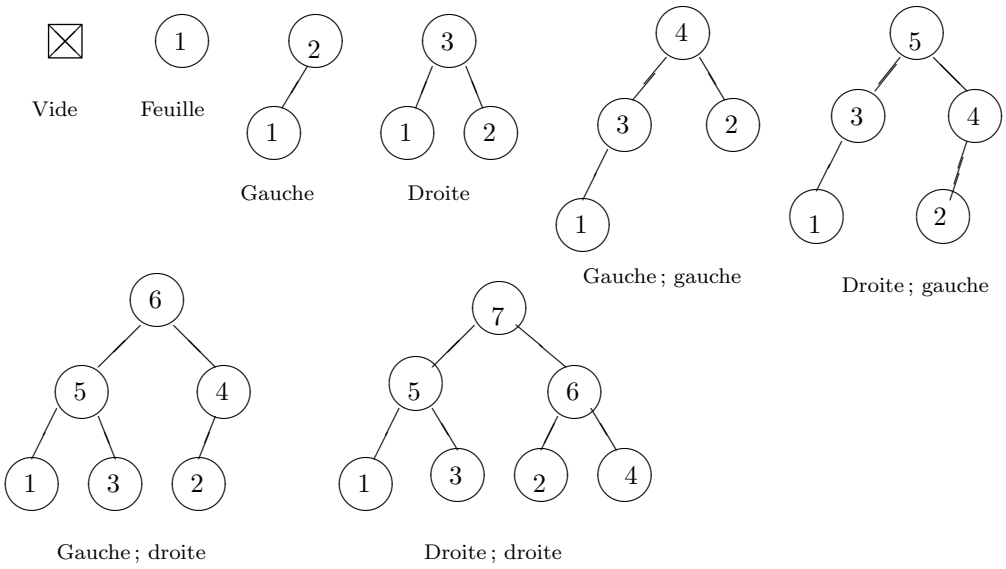
Nota : si R est transitive, critère local (plus opératoire)
– si $t = Br(a_0, t_1, t_2)$ et si, pour $i \in \{1, 2\}$, $t_i = Br(a, t'_1, t'_2)$, on vérifie aRa_0 et t_i est un tas.

Création d'un tas comme arbre binaires de recherche, une fonction d'insertion invariante pour la propriété *"être un tas"*.

Idée : alterner les étapes

1. descendre alternativement à gauche puis à droite en permutant les nœuds pour préserver l'ordre.
2. descendre alternativement à droite puis à gauche en permutant les nœuds pour préserver l'ordre.

Séquence d'insertion, exemple (on ne dessine pas les sous-arbres vides aux feuilles)



Codage : énumération des suites binaires (0 pour gauche et 1 pour droite) : 0, 1, 00, 10, 01, 11, etc.

```
let ins_heap r a h p =
  let rec loop a h m =
    match h with
      Nil -> Br(a, Nil, Nil)
    | Br(a0, h1, h2) -> (
        if ((p land m) = 0) then
          if (r a a0) then
            Br(a, (loop a0 h1 (m lsl 1)), h2)
          else
            Br(a0, (loop a h1 (m lsl 1)), h2)
        else
          if (r a a0) then
            Br(a, h1, (loop a0 h2 (m lsl 1)))
          else
            Br(a0, h1, (loop a h2 (m lsl 1)))
      )
   in
     loop a h 1

let from_list ns =
  let rec loop ns p h =
    match ns with
      [] -> h
    | n::ns -> (loop ns (succ p) (ins_heap (>) n h p))
  in
    loop ns 1 Nil
```

Solution fonctionnelle : descendre toujours à gauche, mais permuter les sous-arbres.

```
let rec ins_heap r a t =
  match t with
    Nil -> Br(a, Nil, Nil)
  | Br(a0, t1, t2) ->
```

```
        if (r a a0) then
          Br(a, (ins_heap r a0 t2), t1)
        else
          Br(a0, (ins_heap r a t2), t1)
```

Suppression de la racine, fusion de deux tas :

```
# let rec heap_merge r h1 h2 =
    match h1, h2 with
      Nil, _ -> h2
    | _, Nil -> h1
    | (Br(a1, h11, h12)), (Br(a2, h21, h22)) ->
      if (r a1 a2) then
        Br(a1, (heap_merge r h11 h12), h2)
      else
        Br(a2, h1, (heap_merge r h21 h22))

  let heap_pop r h =
    match h with
      Nil -> raise Empty
    | Br(a, h1, h2) -> (a, (heap_merge r h1 h2))
  ;;
val heap_merge : ('a -> 'a -> bool) -> 'a btree -> 'a btree -> 'a btree =
  <fun>
val heap_pop : ('a -> 'a -> bool) -> 'a btree -> 'a * 'a btree = <fun>
```

Application : files d'attente avec priorité (exo)

Feuilles étiquetés Type concret :

```
type 'a btree2 =
  Empty
| Lf of 'a
| Br of 'a btree2 * 'a btree2
```

Applications : merge sort, huffmann (exo ?)

Arbres binaires de recherche équilibrés AVL, exemple : implantation (édulcorée) du module `Set`.

Type concret : on mémorise la hauteur de chaque nœud

```
type 'a t = Empty | Node of 'a t * 'a * 'a t * int

let height = function
  Empty -> 0
| Node(_, _, _, h) -> h
```

Création d'un nœud, calcul de la hauteur

```
let create l x r =
  Node(l, x, r, ((max (height l) (height r)) + 1))
```

Si `l` et `r` sont des AVL et que la différence de leur hauteurs n'excède pas 2 alors `(create l x r)` est un AVL.

Création d'un nœud avec rééquilibrage

```
let bal l x r =
  let hl = (height l)
```

```
  and hr = (height r) in
    if (hl > hr + 2) then (
      match l with
        Empty -> (invalid_arg "Set.bal")
      | Node(ll, lv, lr, _) ->
          if (height ll) >= (height lr) then
            (create ll lv (create lr x r))
          else (
            match lr with
              Empty -> (invalid_arg "Set.bal")
            | Node(lrl, lrv, lrr, _)->
                (create (create ll lv lrl) lrv (create lrr x r))
          )
       )
     else if (hr > hl + 2) then (
       match r with
         Empty -> (invalid_arg "Set.bal")
       | Node(rl, rv, rr, _) ->
          if (height rr) >= (height rl) then
            (create (create l x rl) rv rr)
          else (
            match rl with
              Empty -> (invalid_arg "Set.bal")
            | Node(rll, rlv, rlr, _) ->
                (create (create l x rll) rlv (create rlr rv rr))
          )
       )
       else (create l x r)
```

Fonction d'ajout d'un élément : préserve l'invariant *"être un AVL"*

```
let rec add x t =
  match t with
      Empty -> Node(Empty, x, Empty, 1)
    | Node(l, v, r, _) ->
        let c = (compare x v) in
          if (c = 0) then
            t
          else if (c < 0) then
            (bal (add x l) v r)
          else
            (bal l v (add x r))
```

Remarque : on utilise la relation d'ordre générique `compare`.

Exo : module `Map`

8.2 Arbres planaires généraux

Implanter la suite des constructeurs $Br_0, Br_1, \ldots, Br_k, \ldots$; n-uplet d'arbres : forêts

Types concrets mutuellement récursifs : arbres et forêts

```
type 'a t =
  Empty_t
```

```
| Br of 'a * 'a f
and
  Empty_f
| Cons of 'a t * 'a f
```

Fonctions mutuellement récursives, ex : énumération préfixe

```
let rec to_list_prefix t =
  match t with
    Empty_t -> []
  | Br(a,ts) -> a::(iter_to_list_prefix ts)
and iter_to_list_prefix ts =
  match ts with
    Empty_f -> []
  | Cons(t,ts) -> (to_list_prefix t)@(iter_to_list_prefix ts)
```

Noter : la notion de *infix* n'a pas vraiment de sens ici.

Solution alternative : avec des listes

```
type 'a t =
  Empty
| Br of 'a * 'a t list
```

Utilisation d'itérateurs sur les listes

```
let rec to_list_prefix t =
  match t with
    Empty -> []
  | Br(a, ts) -> a::(List.concat (List.map to_list_prefix ts))
```

Autre possibilité : avec des tableaux

```
type 'a t =
  Empty
| Br of 'a * 'a t array
```

Entrée/sorties Les arbres comme S-expressions :

sexp-tree : := atom | () | (atom sexp-trees)
serp-trees : := | sexp-tree sexp-trees

Fonction d'écriture (arbres avec listes de sous-arbres)

```
let write oc write_elt t =
  let write_string = output_string oc in
  let write_char = output_char oc in
  let write_elt = write_elt oc in
  let rec loop t =
    match t with
        Empty -> write_string "()"
      | Br(n, []) -> (
          write_char '(';
          write_elt n;
          write_char ')'
        )
      | Br(n, ts) -> (
          write_char '(';
```

```
            write_elt n;
            List.iter (fun t -> write_char ' '; loop t) ts;
            write_char ')'
          )
    in
      loop t
```

Pour lire un arbre
- lire un atome, si ok sortir ;
- sinon
  - lire une ouvrante ;
  - lire une fermante, si ok, sortir ;
  - sinon,
    - lire un atom ;
    - lire une suite d'arbres
    - lire une fermante

Pour lire une suite d'arbres
- lire un arbre, si ok lire une suite d'arbres
- sinon, sortir

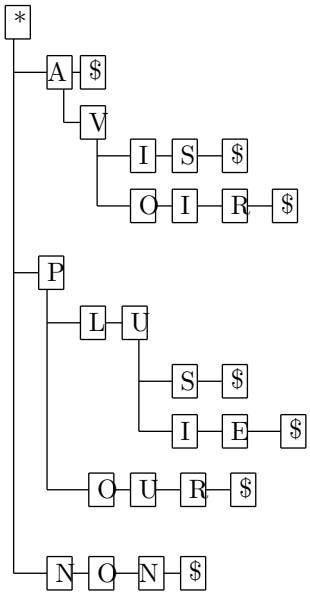
Fonction de lecture (utilise `read_c`, `read_s`, etc. de 2.4)

```
let syntax_error msg ic =
  failwith (Printf.sprintf "read_sexp: syntax error [%s] at %d"
                           msg (pos_in ic))

let read_sexp_as_tree fname =
  let ic = open_in fname in
  let rec loop () =
    skip_seps ic;
    try
      Br(read_atom ic, [])
    with No_reading -> (
      let lpar = read_lpar ic in
        try
          let rpar = read_rpar ic in Empty
        with No_reading -> (
          let atom1 = read_atom ic in
              let exps = it_loop() in
              let rpar = read_rpar ic in
                Br(atom1, exps)
              syntax_error "" ic*)
            )
       )
  and it_loop () =
    try
      let exp = loop() in
        exp::(it_loop())
    with No_reading | End_of_file ->
      []
  in
    try
      let exp = loop() in
        close_in ic;
```

```
    exp
with No_reading ->
  syntax_error "" ic
```

Applications les *tries*, arbres lexicaux, ensemble de mots (lexique) : partage des préfixe

Exemple :



L'* à la racine marque le début des mots, les $ aux feuilles marquent les fins de mots. Un mot $C_1...C_k$ apparatient à l'arbre s'il existe une branche $*C_1...C_k\$$. L'arbre contient : A, AVIS, AVOIR, PLUS, PLUIE, POUR, et NON. L'arbre ne contient pas POU.

Interface (du module `Trie`)

```
(** Arbres lexicaux, "tries" *)

type t
(** Abstract type of tries *)

val create : unit -> t
(** Create a new empty trie *)

val search : string -> t -> bool
(** [search s t] return [true] iff [s] belongs to [t] *)

val add : string -> t -> t
(** [add s t] return a trie where s has been added to [t] *)

val rem : string -> t -> t
(** [rem s t] return a trie where s has been removed from [t] *)

val to_list : t -> string list
(** Return the list of words contained in the given trie *)
```

Implantation : on utilise la repésentation des arbres généraux avec liste de sous-arbres. Les étiquettes des nœuds sont des caractères.

Constructeur :

```
let create () = Br('*', [])
```

Noter : l'intérêt du type abstrait, on est sûr d'avoir une * à la racine.

Liste des mots : cas `Br(c, ts)`

1. créer récursivement la liste des listes de mots de chaque `t` dans `ts` (`map to_list`) ;
2. concaténer ces listes pour obtenir une liste de mots (`concat`) ;
3. pour chaque mot, ajouter le caractère `c` (`map (fun s -> (Char.escaped c)^s)`) .

```
let rec to_list t =
  match t with
      Empty -> []
    | Br(c, []) -> [Char.escaped c]
    | Br(c, ts) ->
        (List.map (fun s -> (Char.escaped c)^s)
           (List.concat (List.map to_list ts)))
```

Auxiliaires : arbre de racine `x` dans une liste `ts`

```
let br_assoc x ts =
  let found x t =
    match t with
        Empty -> false
      | Br(y, _) -> (x = y)
  in
    List.find (found x) ts

let br_mem_assoc x ts =
  try ignore(br_assoc x ts); true with Not_found -> false
```

Recherche d'un mot

```
let search w t =
  let len = String.length w in
  let rec loop i ts =
    if (i = len) then
      (br_mem_assoc '$' ts)
    else
      match (br_assoc w.[i] ts) with
          Empty -> raise Not_found
        | Br(_, ts) -> (loop (succ i) ts)
  in
    match t with
        Br('*', ts) -> (try loop 0 ts with Not_found -> false)
      | _ -> invalid_arg "search"
```

Auxiliaire : `br_get : char -> t list -> t * t list`

```
(** br\_get: char -> t list -> t * t list
    (br_get x ts) donne le couple (t,ts') avec t de racine x et
    ts' est ts priv\'ee de t. Renvoie (Empty, ts) si pas trouve. *)
let br_get x ts =
  let rec loop ts ts' =
```

```
    match ts with
        [] -> (Empty, ts')
      | Empty::ts -> loop ts (Empty::ts')
      | (Br(y,ts1))::ts ->
          if (x = y) then
            (Br(y,ts1), ts@ts')
          else
            loop ts ((Br(y,ts1))::ts')
  in
    loop ts []
```

Noter : utilisation de `Empty` qui ne correspond jamais à un trie.

Suppression d'un mot : supprimer une branche

```
let rem w t =
  let len = String.length w in
  let rec loop i ts =
    if (i = len) then (
      match (br_get '$' ts) with
          Empty, _ -> raise Not_found
        | _, ts' -> ts'
    )
    else
      match (br_get w.[i] ts) with
          Empty, _ -> raise Not_found
        | Br(c, ts1), ts' -> (
            match (loop (succ i) ts1) with
                [] -> ts'
              | ts1' -> (Br(c, ts1'))::ts'
          )
  in
    match t with
        Br('*', ts) -> Br('*', (loop 0 ts))
      | _ -> invalid_arg "search"
```

Auxiliaire : crée un arbre (linéaire) à partir d'un mot

```
let init w =
  let len = String.length w in
  let rec loop i =
    if (i = len) then
      Br('$',[])
    else
      Br(w.[i],[loop (succ i)])
  in
    loop 0
```

Ajout d'un mot :

```
let add w t =
  let len = String.length w in
  let rec loop i ts =
    if (i = len) then
      if not (br_mem_assoc '$' ts) then
        (Br('$',[]))::ts
      else
```

```
            ts
          else
            match (br_get w.[i] ts) with
                Empty, ts -> (init (String.sub w i (len - i)))::ts
              | Br(x, ts1), ts -> (Br(x,(loop (succ i) ts1)))::ts
        in
          match t with
              Br('*', ts) -> Br('*', (loop 0 ts))
            | _ -> invalid_arg "add"
```

TD/TP : arborescence des répertoires

9 Graphes

Ensemble de *sommets* + relation binaire sur les sommets, *liaisons*.

Graphe *orienté* (liaison : *arcs*) ou non (liaison : *arêtes*, relation symétrique)

Sommets reliés : *adjacents*, *successeurs/prédécésseur* si orienté.

Chemin : suite $s_1..s_k$ tels que s_i, s_{i+1} reliés.

Cycle (ou *circuit*) : chemin $s_0..s_0$.

Graphe *connexe* : pour tous sommets s, s' il existe un chemin $s..s'$

Sous-graphe : sous-ensemble de sommets et sous-ensemble de liaisons pour ces sommets.

Sommets étiquetés ; liaisons étiquetées, *coût*, graphe *pondéré*.

Type abstrait :

```
(** Graphs
  Oriented, vertices (nodes) labelled with ['a]
*)

type 'a t
(** Abstract type for graphs *)

val create: unit -> 'a t
(** [create()] creates a new empty graph *)

val add_node: 'a t -> 'a -> 'a t
(** [add_node g n] return the graph where the node [n] has been added to
  the graph [g]. *)

val rem_node: 'a t -> 'a -> 'a t
(** [rem_node g n] return the graph where the node [n] has been removed from
  the graph [g]. Raise [Failure "rem_node"] if (n,n') or (n',n) is an
  edge of [g] for some n'. *)

val nodes_of: 'a t -> 'a list
(** [nodes_of g] give the list of nodes of graph [g] *)

val add_edge: 'a t -> 'a -> 'a -> 'a t
(** [add_edge g n1 n2] return the graph where the edge ([n1],[n2]) has been added
  to the graph [g]. Raise [Not_found] if [n1] or [n2] are not nodes
  of [g] *)

val rem_edge: 'a t -> 'a -> 'a -> 'a t
(** [add_edge g n1 n2] return the graph where the edge ([n1],[n2]) has been
  removed from [g]. *)

val has_edge: 'a t -> 'a -> 'a -> bool
(** [has_edge g n1 n2] return [true] iff [g] contains the edge
  ([n1],[n2]). *)

val adjs_of: 'a t -> 'a -> 'a list
(** [adjs_of g n] return the list of adjacents to [n] in [g].
```

```
      Raise [Not_found] when [n] is not a node of [g]. *)
```

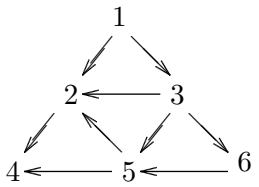
Fonction de création d'un graphe à partir d'une liste de couples

```
let graph_of_list nns =
let force_add g (n1, n2) =
  add_edge (add_node (add_node g n1) n2) n1 n2
```

Affichage d'un graphe sous forme de listes d'ajacences

```
let print_as_adj print_node g =
  let print_n_adjs n =
    print_node n;
    print_char ':';
    List.iter (fun n -> print_char ' '; print_node n) (adjs_of g n)
  in
    List.iter (fun n -> print_n_adjs n; print_newline()) (nodes_of g)
```

Exemple, les étiquettes sont des entiers




```
# let nns =
    [1,2; 1,3; 2,4; 3,2; 3,5; 3,6; 5,2; 5,4; 6,5]
  in
    print_as_adj print_int (graph_of_list nns) ;;
6: 5
5: 4 2
4:
3: 6 5 2
2: 4
1: 3 2
- : unit = ()
```

Recherche d'un chemin : attention aux cycle, marquer les sommets visités.

Algorithme récursif, exploration *en profondeur* ; trouver n' en partant de n, fonction $FindPath(n)$

$FindPath(n) =$
  SI $n = n'$ ALORS
    Ok
  SINON
    marquer n
    REPETER
      Choisir p adjacent à n non marqué
      Essayer $FindPath(p)$,
    JUSQU'A Ok ou Plus d'adjacent non marqué
    SI Plus d'adjacent non marqué ALORS
      Echec

```
let path n1 n2 g =
  let m = ref [] in
```

```
let mark n = m := n::!m in
let is_marked n = (List.mem n !m) in
let rec loop n p =
  if (n = n2) then
    n::p
  else (
    mark n;
    (loops (adjs_of g n) (n::p))
  )
and loops ns p =
  match ns with
      [] -> raise Not_found
    | n::ns -> (
        if (is_marked n) then
          (loops ns p)
        else (
          try loop n p
          with Not_found -> loops ns p
        )
      )
in
  loop n1 []
```

NB : le liste résultat contient le chemin à l'envers.

Tri topologique, ordre partiel ; détection de cycle. Liste topologique des nœud $n_1, .., n_n$ tel que pour toute liaison (n_i, n_j), $i < j$.

Degré (entrant) d'un nœud : nombre de ses prédécesseurs dans le graphe.

Algo : D table des degrés des noeuds,

REPETER
  Choisir n non marqué tel que $D(n) = 0$
  SI pas de tel n ALORS
    Sortir
  SINON
    Marquer n
    POUR TOUT n' (non marqué) adjacent à n
      Décrémenter $D(n')$
JUSQU'A Sortir
SI il existe n non marqué tel que $D(n) > 0$ ALORS
  Echec

Codage : D, liste d'association ; utiliser un degré négatif comme marquage.

Fonction de choix : retourne un noeude de degré minimal pour l'ordre $0 < 1 < 2 < .. < -1 = -2..$ (négatif $\approx$ infini).

```
let find_deg_0 nds =
  let deg_min nd1 nd2 =
    let _,d1 = nd1
    and _,d2 = nd2 in
      if !d1 < 0 then nd2
      else if !d2 < 0 then nd1
```

```
      else if !d1 < !d2 then nd1
      else nd2
  in
    match nds with
      nd::nds -> List.fold_left deg_min nd nds
    | _ -> invalid_arg "find_deg_0"
```

Noter : si résultat négatif, tous les sommets sont marqués ; si résultat positif, il existe un sommet non marqué (de debré non nul).

Liste topologique (à l'envers)

```
let decr_pos_deg nds n =
  let d = List.assoc n nds in
    if !d > 0 then decr d

let topo g =
  let nds = degs g in
  let decr_deg n = decr (List.assoc n nds) in
  let rec loop ns =
    let n,d = find_deg_0 nds in
      if !d < 0 then ns
      else if !d > 0 then failwith "topo: cycle"
      else (
        decr d;
        List.iter decr_deg (adjs_of g n);
        (loop (n::ns))
      )
  in
    loop []
```

Le plus court chemin d'un noeud n_0 à tous les accessibles : Dijkstra.

Algo : $D(n')$ distance de n à n'. Initialisation, puis mise à jour par parcours en profondeur.

Soit ns l'ensemble des noeuds, moins n_0

POUR TOUT noeud $n \in ns$
  SI la liaison (n_0, n) existe ALORS
    $D(n) \leftarrow 1$
  SINON
    $D(n) \leftarrow \infty$
REPETER
  Choisir $n \in ns$ tel que $D(n)$ minimal
  POUR TOUT n' adjacent à n
    SI $D(n) + 1 < D(n')$ ALORS
      $D(n') \leftarrow D(n) + 1$
  $ns \leftarrow ns - n$
JUSQU'A $ns = \emptyset$

Codage : toutes les distances sont positives, négatif = infini. Comparaison des distances

```
let comp_dist n1 n2 =
  if n1 < 0 then
    if n2 < 0 then 0 else 1
  else
    if n2 < 0 then -1 else compare n1 n2
```

Noeuds quelconques (pas nécessairement entiers), table de hachage pour D. Pour mémoriser le chemin, mémoriser le prédécesseur.
Type et initialisation de la table des distances (et prdécésseurs)

```
type 'a info =
    {
      mutable dist : int;
      mutable pred : 'a option
    }

let mk_dist g n0 ns =
  let dtab = Hashtbl.create (succ (List.length ns)) in
  let init n =
    Hashtbl.add dtab n
      (if (has_edge g n0 n) then
         { dist = 1; pred = Some n0 }
       else
         { dist = -1; pred = None })
  in
    Hashtbl.add dtab n0 { dist = 0; pred = None };
    List.iter init ns;
    dtab
```

Fonction pour Dijkstra, renvoie la table.

```
let dijk g n0 =
  let ns = list_rem n0 (nodes_of g) in
  let dtab = mk_dist g n0 ns in
  let comp_info n1 n2 =
    comp_dist
      (Hashtbl.find dtab n1).dist
      (Hashtbl.find dtab n2).dist
  in
  let update n n' =
    let i = Hashtbl.find dtab n in
    let i' = Hashtbl.find dtab n' in
    let d = i.dist + 1 in
      if (comp_dist d i'.dist) < 0 then (
        i'.dist <- d;
        i'.pred <- Some n
      )
  in
  let rec loop ns =
    match ns with
        [] -> ()
      | _ -> (
          let n = find_min comp_info ns in
          let ns = list_rem n ns in
            List.iter (update n) (adjs_of g n);
            loop ns
        )
  in
    loop ns;
    dtab
```

Test et affichage (noeuds entiers)

```
# let print_info n i =
    let spred = match i.pred with None -> "*" | Some n -> string_of_int n in
    let sdist = if i.dist < 0 then "*" else (string_of_int i.dist) in
      Printf.printf" %d: dist = %s, pred = %s\n" n sdist spred
val print_info : int -> int info -> unit = <fun>
# print_as_adj print_int g ;;
6: 5
5: 4 2
4:
3: 6 2
2: 4
1: 2 3
- : unit = ()
# Hashtbl.iter print_info (dijk g 1) ;;
 5: dist = 3, pred = 6
 6: dist = 2, pred = 3
 1: dist = 0, pred = *
 2: dist = 1, pred = 1
 3: dist = 1, pred = 1
 4: dist = 2, pred = 2
```

Table des matières