Introduction à la programmation fonctionnelle en Ocaml

Maria Virginia Aponte

CNAM-Paris

Année 2010/2011

Pourquoi Ocaml?

- C'est un langage *multi-paradigme*, concis et puissant:
  - traits fonctionnels, et *impératifs*,
  - système de *modules* génériques,
  - Programmation *objet* sophistiquée,
  - nombreuses librairies
- Il permet de programmer aisément des applications complexes;
- C'est un langage sûr et facile à employer:
  typage *fort* et *inférence des types.*

Pourquoi Ocaml? (suite)

- Son noyau fonctionnel est bien adapté à l'*analyse mathématique des programmes*;
- C'est langage qui possède deux modes:
  - *compilé* (*byte-code* ou code natif), avec performances comparables à C;
  - et un *mode interactif*, qui facilite les tests.
- *Gestion mémoire automatique* (comme en Java).

Documentation, distribution et plus de détails:
`http://caml.inria.fr/ocaml`.

Un peu d'histoire

Ocaml est un langage de la famille ML, développé depuis $\approx$ 1980.

1975 : Robin Milner propose ML comme métalangage (langage de script) pour l'assistant de preuve LCF. Devient rapidement un langage de programmation à part entière.

1985 : Développement de Caml à l'INRIA, de Standard ML (Edinburgh), de "SML of New-Jersey", de Lazy ML(Chalmers), de Haskell (Glasgow), etc.

990-95 : Implantation de Caml-Ligh, compilateur vers du code natif + système de modules.

1996 : Objets et classes (Ocaml).

001-... : Arguments étiquetés, Méthodes polymorphes, librairies partagées, Modules récursifs, private types, ...

Domaines d'utilisation

Un langage multi-paradigme et d'usage général avec des **domaines de prédilection**:

- Calcul symbolique: Preuves mathématiques, compilation, interprétation, analyse de programmes.
- Prototypage rapide: Langage de script; Langages dédiés.
- Programmation distribuée (bytecode rapide).

Utilisé en enseignement et recherche:

- Classes préparatoires.
- Utilisé dans de grandes universités (Europe, US, Japon, etc.).

Utilisation en industrie et grands logiciels

- en Industrie: Startups, CEA, EDF, France Telecom, Simulog, Airbus...
- applications en Industrie: aéronautique, nucléaire, informatique financière, bio-informatique, développement web, ...
- des Gros Logiciels: Coq, Ensemble, ASTREE, (voir la bosse d'OCaml)

Des outils système:

- Unison (synchronisation de fichiers, distribution Linux)
- MLdonkey (peer-to-peer, distrib. Linux),
- Libre cours (Site WEB),
- Lindows (outils système pour une distribution de Linux)

La programmation fonctionnelle

Une fonction relie les éléments de deux ensembles:

- le *domaine* et le *co-domaine* de la fonction;
- A chaque valeur du *domaine* correspond *une unique* valeur de son *co-domaine*.

$$\begin{array}{lccl} tri: & \text{suite d'articles} & \longrightarrow & \text{suite d'articles triés} \\ & s & \longmapsto & tri(s) \end{array}$$

Caractéristique essentielle des fonctions:
comportement *stable et indépendante du contexte* $\Rightarrow$ une fonction appliquée sur un même argument, donne toujours le même résultat.

Opérations qui ne sont pas des fonctions

```
int i = 0;

int f (int j) {
  i = i+j;
  return i;
}
```

L'opération `f` ne correspond pas à une fonction:

- résultat qui n'est pas seulement fonction de son argument,
- mais aussi de la valeur courante de la variable globale `i`,
- qui de plus est modifiée lors de chaque appel.
- Le comportement de la fonction *dépend de l'état des variables au moment de l'appel*,
- `f` ne peut pas être étudiée en dehors de son contexte.

Principes de la PF: (1) Favoriser les fonctions

L'idée derrière la programmation fonctionnelle est de

favoriser la réalisation des calculs à l'aide de fonctions au sens mathématique du terme.

. . . afin de tirer parti de leur séparation du contexte et de la stabilité qui en découle.

Principe (2): Les fonctions comme des valeurs

En programmation fonctionnelle les fonctions sont des valeurs comme les autres. On peut:

- les stocker dans une variable,
- les passer en paramètre, les donner en résultat

On parle alors de *pleine fonctionnalité* ou *ordre supérieur* ou encore des fonctions comme étant des entités de *première classe*.

Généralisation avec fonctions en paramètre

Considérons une fonction de tri d'un ensemble d'articles:

- dépend du critère d'ordonnancement pour réaliser le tri;
- Plutôt que de le fixer nous le passons en paramètre.
- La fonction de tri devient *générale à tout critère*,
- et peut être employée dans beaucoup de contextes.

$$\begin{array}{lcccl} tri: & \text{critère de comparaison} & \times & \text{suite d'articles} & \rightarrow & \text{suite d'articles triés} \\ & (f_{comp}, s) & & & \mapsto & tri(f_{comp}, s) \end{array}$$

En Ocaml:

```
let tri(f_compare, articles) = ...
```

Principe (3): La généricité par les types

Généricité: *ne pas restreindre inutilement* le type des données ou fonctions. Les rendre génériques à tout type de valeur...

Utilise: paramètres de type. Comme en Java 1.5:

```
class Cellule<A> {A contenu; Cellule suivant};
```

ou en C++ (*templates*) :

```
template <typename A>
A identity (A x) {
  return x;
}
```

En Ocaml ce mécanisme est incorporé automatiquement: les paramètres de type sont inférés par le typeur.

Avantages de la PF + généricité

1. Fonctions au sens mathématique $\approx$ *boîtes noires* $\Rightarrow$ conception +facile, +modulaire, débogagge +facile.
2. Toutes les valeurs de première classe $\Rightarrow$ uniformité, facilité de programmation et de structuration.
3. Généralisation des fonctions avec paramètres fonctionnels $\Rightarrow$ favorise la structuration et la réutilisation.
4. Formalisme mathématique sous-jacent: favorise la vérification $\Rightarrow$ +grande sécurité de programmation.
5. Généricité par les types + typage fort $\Rightarrow$ allie la flexibilité du typage et réutilisation avec la sécurité du typage $\Rightarrow$ rapidité, réutilisation, sécurité.

Contraintes de la PF

- Pas d'effets de bords: pas d'affectation, pas de variables modifiables. Seules les constantes sont admises;
- Une fonction ne peut pas changer ce qui lui est extérieur; Le comportement d'une fonction ne peut pas être modifié de l'extérieur;
- Constructions proscrites: boucles itératives, *goto*.
- La récursivité est à utiliser afin de coder les boucles.
- La gestion implicite de la mémoire devient indispensable.
- Style déroutant pour les novices.

Le mode compilé

Éditer le source

```
hello.ml
print_string  "Hello World!\n";;
```

Compiler, lier et exécuter (sous le Shell d'Unix):

```
ocamlc -o hello hello.ml
./hello
Hello World!
```

Le mode interactif

Dans ce mode Ocaml analyse et répond à chaque phrase entrée par l'utilisateur. Pour lancer ce mode on tape la commande `ocaml` (sous le shell Unix):

```
% ocaml
        Objective Caml version 3.06

#
```

Le caractère `#` invite l'utilisateur à entrer une phrase écrite dans la syntaxe Ocaml (chaque phrase Ocaml doit terminer par `;;` puis l'utilisateur valide sa frappe par un retour chariot).

Le mode interactif(suite)

- Ocaml analyse chaque phrase:
  - calcule son type (*inférence des types*),
  - la traduit en langage exécutable (*compilation*)
  - et enfin l'*exécute* afin de fournir la réponse demandée.

- La réponse donnée en mode interactif contient:
  - Le nom de la variable déclarée s'il y en a.
  - Le type trouvé pendant le typage.
  - La valeur calculée après exécution.

Exemple

```
# let x = 4+2;;
val x : int = 6
```

La réponse d'Ocaml signale :

- l'identificateur `x` est déclaré `(val x)`,
- avec le type des entiers `(:int)`,
- et la valeur 6 `(=6)`.

L'identificateur `x` est lié à la valeur 6 de type `int` dans l'environnement global. Il peut être employé plus tard:

```
# x;;
val x : int = 6

# x * 3;;
- : int = 18
```

Les types

- Types de *base prédéfinis*;
- Types *composites* prédéfinis ou à déclarer (certains étant modifiables, d'autres constants):
  - tableaux (prédéfini, modifiable)
  - chaînes de caractères (prédéfini, modifiable),
  - n-uplets (prédéfini, modifiable),
  - enregistrements (à déclarer, constant ou modifiable),
  - types inductifs (à déclarer, constant ou modifiable)
  - listes (prédéfini, constant)

Les types de base

Type	Constantes	Primitives
`unit`	`()`	pas d'opération!
`bool`	`true false`	`&& || not`
`char`	`'a'  '\n'  '\097'`	`code, chr`
`int`	`1  2  3`	`+  -  *  /  max_int`
`float`	`1.0  2.  3.14`	`+.  -.  *.  /.` **cos**
`string`	`"a\tb\010c\n"`	`^  s.[i]  s.[i] <- c`

Comparaison (pour tous les types) `=, > , < , >=, <=, <>.`

Exemples (expressions)

```
# 1+ 2;;
- : int = 3

# 1.5 +. 2.3;;
- : float = 3.8

# let x = "cou" in x^x;;
- : string = "coucou"

# 2 > 7;;
- : bool = false

# "bonjour" > "bon";;
- : bool = true
```

Les types composites

Types composites	Caractéristiques	Exemple
`int array`	prédéfini, modifiable	`[<1;2;3>]`
`string`	prédéfini, modifiable	`"bonjour"`
`int*string*bool`	prédéfini	`(4,"abc",true)`
enregistrement	à déclarer	`{nom="toto";num=45}`
inductif	à déclarer	`Noeud(1, Vide, Vide)`
`char list`	prédéfinis	`['a'; 'b'; '7']`
objets	à déclarer	non présenté

Syntaxe des programmes

Un programme est une suite de phrases:

définition de valeur	`let` x = e
définition de fonction	`let` f x1 ... xn = e
définition de fonctions	`let [ rec ]` f1 x1 ... = e1 ...
(mutuellement récursives)	`[ and` fn xn ... = en]
définition de type(s)	`type` q1 = t1... [`and` qn = tn]
expression	e

Les phrases se terminent par ;; optionnel entre des déclarations, mais obligatoires sinon.

Exemple de programme

Déclarer une valeur (une donnée):

```
# let baguette = 4.20;;
val baguette : float = 4.2
```

Déclarer une fonction pour résoudre un problème:

```
# let euro x = x /. 6.55957;;
val euro : float -> float = <fun>
```

Appliquer la fonction aux données pour obtenir la solution:

```
# euro baguette;;
- : float = 0.64028587239712364 5
```

Les expressions

Construction permettant de réaliser un calcul *(valeur résultat)*.

- Constituée soit d'une constante, soit d'opérations et/ou appels de fonctions.
- On *évalue* une expression afin d'obtenir son résultat.
- Toute expression
  - *retourne une valeur résultat*,
  - *a un type*: celui de sa valeur résultat.

Exemples d'expressions

Expressions avec leurs valeurs et types, en supposant:

- `x` =-2,
- `f` est la fonction qui teste si son argument est positif.

expression	valeur	type
5	5	`int`
`3 + x`	1	`int`
`f(3)`	true	`bool`
`f(x) || x=7`	false	`bool`
`if x>0 then 1 else 2`	2	`int`

Syntaxe des expressions

définition locale	`let` x = e1 `in` e2
fonction anonyme	`fun` x1 ... xn `->` e
appel de fonction	f e1 ... en
variable	`x`
	(`M.x` si x est défini dans le module M)
valeur construite	(e1, e2)
(dont les constantes)	1, 'c', "aa"
analyse par cas	`match` e `with` p1 `->` e1 ... `|` pn `->` en
boucle for	`for` i = e0 `to` ef `do` e `done`
boucle while	`while` e0 `do` e `done`
conditionnelle	`if` e1 `then` e2 `else` e3
une séquence	e; e '
parenthèses	(e) `begin` e `end`

Déclarations globales

Un identificateur est déclaré *globalement* avec `let`:

```
# let x = 7;;
val x : int = 7
```

Il est ajouté dans l'*environnement global*, et pourra être employé plus tard.

```
# x+2;;
- : int = 9
```

Il est *constant* (non modifiable).

Déclarations locales = expressions

C'est une expression qui utilise des identificateurs déclarés *localement à l'expression* avec la syntaxe:

`let` *id*= *valeur* `in` *expression*

```
#let y = 5 in 3*y;
 - : int =15
#y;;
Unbound value y
```

$\Rightarrow$ identificateur déclaré visible uniquement après le `in` (local à l'expression).

Cette construction *est une expression* et retourne donc une valeur!

Déclarations globales et locales

Différence importante:

- une déclaration globale est une *déclaration* et ne retourne pas de valeur;
- une déclaration locale est une *expression* et retourne donc une valeur.

```
# 2 + (let y = 5 in 3 *y);;
- : int = 17

# 2 + (let y = 15);;
Syntax error
```

Valeurs constantes et masquage

Les identificateur déclarés jusqu'ici sont **constants**:

- La déclaration `let x =`v ne permet la modification de la valeur v associée à `x`
- On pourra introduire une nouvelle liaison pour `x`, `let x =` v_1 qui masque la première dans la suite du programme, mais si "l'ancien" `x` est utilisé ailleurs, sa valeur v est toujours d'actualité.

Expression conditionnelle

- a toujours deux expressions-branches qui nécessairement *de même type*;
- son résultat est celui de l'une des deux branches.

```
# let b = if 2<3 then "vrai" else "faux";;
val b : string = "vrai"

# if 7>2 then 1 else true;;
Error:   if x>2 then 1 else true;;
                            ^^^^
This expression has type bool but is here used
 with type int
```

Déclarer une fonction

Un identificateur de fonction est déclaré par un `let`.

Ocaml nous donne son type (avec une $\rightarrow$), et sa valeur (toujours `<fun>`).

```
# let double y = y*2;;
val double : int -> int = < fun >
```

Une syntaxe alternative:

```
# let double (y) = y*2;;
val double : int -> int = < fun >
```

La première est la syntaxe usuelle en Ocaml.

Le type d'une fonction à un argument

$$\texttt{t} \rightarrow \texttt{q}$$

- `t` est le type de l'argument de la fonction,
- `q` celui du résultat de la fonction.

```
# let double y  =   y*2;;
val double : int -> int = < fun >
```

Ici, `l'argument (y)` et le `résultat (y*2)` sont de type `int`.

L'identificateur `double` est désormais lié à une *valeur fonctionnelle*:

```
# double;;
val double : int -> int = < fun >
```

Application de fonctions

On applique une fonction en la faisant suivre de son argument (éventuellement entre parenthèses). Elle retourne un résultat:

```
# double 9;;
- : int = 18

# double(9);;
- : int = 18
```

L'application peut également se faire avec deux syntaxes: avec ou sans parenthèses.

Déclarations vs expressions

- Une déclaration (globale):
  - sert à introduire un identificateur pour une donnée simple ou pour une fonction;
  - ne retourne pas de résultat mais enrichit l'environnement global des déclarations;
- Une expression:
  - sert à réaliser un calcul et retourne son résultat;
  - ne change pas l'environnement des déclarations;
  - peut contenir des déclarations locales.

Résumé général

- Il n'y a ni instructions, ni procédures, ni variables modifiables.
- Toute valeur est constante.
- Tout sous-programme est une fonction.
- Toute expression retourne une valeur et a un type.
- En dehors des déclarations toute phrase Ocaml est une expression et correspond ainsi au calcul d'une valeur.
- Les fonctions sont des valeurs comme les autres (valeurs de première classe).
- Les types sont inférés.

Fonctions à plusieurs arguments

Deux syntaxes possibles:

- arguments "un-par-un" séparés par des espaces;
- arguments "enfermés" dans un n-uplet.

Arguments un-par-un:

```
#  let moyenne x y = (x+.y)/. 2.0;;
val moyenne : float -> float -> float = <fun>
```

Arguments dans un n-uplet:

```
#  let moyenne (x, y) = (x+.y)/. 2.0;;
val moyenne : float*float -> float = <fun>
```

Notez que les types sont différents!

Fonctions avec arguments un-par-un

Les arguments sont *séparés par des espaces* dans l'entête de la fonction:

```
#  let moyenne x y = (x+.y)/. 2.0;;
val moyenne : float -> float -> float = <fun>
```

Le type d'une fonction à deux arguments est de la forme:

$$t_1 \rightarrow t_2 \rightarrow q$$

où

- t_1 et t_2 sont les types des deux arguments,
- q est le type du résultat.

Type des fonctions un-par-un

La fonction:

$$\texttt{let f } x_1\ x_2 \ldots\ x_n = corps$$

possède n arguments $x_1, x_2, \ldots\ x_n$ et un résultat calculé par *corps*.
Son type est de la forme:

$$t_1 \rightarrow t_2 \ldots \rightarrow t_n \rightarrow t_{corps}$$

où t_i est le type de l'argument x_i, et t_{corps} est le type du résultat calculé par *corps*.

Appliquer une fonction un-par-un

Pour appliquer une fonction à plusieurs arguments, "un-par-un" on lui passe ses arguments, les uns après les autres, séparés par des espaces:

```
# let mot_compose x y = x^"-"^y;;
val mot_compose : string -> string -> string = <fun>

# mot_compose "mot" "cle";;
- : string = "mot-cle"

# let somme x y = x + y;;
val somme : int -> int -> int = <fun>

# somme 3 4;;
- : int = 7
```

N-uplets

N-uplet: Agrégat de n valeurs $v_1, v_2, \ldots v_n$ separées par des virgules.

Exemples: (1,true) $\Rightarrow$ 2-uplet; ("bonjour", 5, 'c') $\Rightarrow$ triplet.

```
# let a = (1, true);;
val a : int * bool = (1, true)

# let livre = ("Paroles", "Prevert, Jacques", 1932);;
val livre : string * string * int =("Paroles",
                    "Prevert, Jacques", 1932)
```

N-uplets (suite)

Un n-uplet permet de mettre dans un “paquet” autant de valeurs que l’on veut. Cela est pratique, si une fonction doit renvoyer “plusieurs” résultats:

```
# let division_euclidienne x y = (x/y, x mod y);;
val division_euclidienne : int -> int -> int * int = <fun>

# division_euclidienne 5 2;;
- : int * int = (2, 1)
```

Type des n-uplets

Le type d'un n-uplet $(v_1, v_2, \ldots, v_n)$ est

$$t_1 * t_2 \ldots * t_n$$

où t_i est le type de la composante v_i.

```
# let a = (1, true);;
val a : int * bool = (1, true)

# let b = ("bonjour", 5, 'c');;
val b : string * int * char = ("bonjour", 5, 'c')
```

Fonctions avec n-uplet d'arguments

Une autre manière de définir une fonction à plusieurs arguments est de les mettre tous dans un n-uplet.

```
# let somme (x,y) = x+y;;
 val somme : int * int -> int = <fun>
```

Attention: cette fonction ne possède *qu'un seul* argument, qui *est ici une paire*. Parmi les appels suivants, le premier est correct, alors que le deuxième ne l'est pas:

```
# somme (2,3);;
- : int = 5

# somme 2 3;;
This function is applied to too many arguments
```

Type de fonctions avec n-uplet d'arguments

La fonction:

$$\texttt{let f}(x_1, x_2, \ldots, x_n) = \mathit{corps}$$

possède 1 argument, qui est un n-uplet de valeurs $(x_1, x_2, \ldots\ x_n)$ et un résultat calculé par *corps*.

Le type du n-uplet $(x_1, x_2, \ldots, x_n)$ est $t_1 * t_2 \ldots * t_n$,
où t_i est le type de chaque x_i. Donc, le type de f est:

$$t_1 * t_2 \ldots * t_n \rightarrow t_{corps}$$

où t_{corps} est le type du résultat calculé par *corps*.

Fonctions à plusieurs arguments

Les fonctions qui prennent leurs arguments un à un, telle:

$$\texttt{let f } x_1 \; x_2 \ldots x_n = \mathit{corps}$$

et celles qui les prennent dans un n-uplet, telle:

$$\texttt{let f } (x_1, x_2, \ldots, x_n) = \mathit{corps}$$

ne sont pas équivalentes: le type de leurs arguments sont différents.

Fonctions à plusieurs arguments

Le type de l'argument d'une fonction renseigne sur la nature des arguments qu'on doit lui passer:

- Pour une fonction avec arguments un-par-un, on doit passer n arguments les uns après les autres. Son type est:

$$t_1 \rightarrow t_2 \ldots \rightarrow t_n \rightarrow t_{res}$$

- Pour une fonction avec un n-uplet d'arguments, on passe un seul argument n-uplet. Son type est:

$$t_1 * t_2 \ldots * t_n \rightarrow t_{res}$$

Définition locale de fonction

Une fonction *globale* peut définir localement une autre fonction *auxiliaire, plus simple*.

Exemple: $x^4 = (x^2)^2$. La fonction `puissance4` est définie en termes de la fonction locale `carre`:

```
let puissance4 x =
   let carre y = y*y        (* definition locale *)
   in carre (carre x);;
val puissance4 : int -> int = <fun>

puissance4 2;;
- : int = 16
```

ligne 3: appel à la fonction locale.

Conventions syntaxiques pour les fonctions

Deux écritures équivalentes pour une fonction:

$$\left.\begin{array}{l} \texttt{let f x = e} \\ \\ \texttt{let f (x) = e} \end{array}\right\} : \quad t_x \rightarrow t_e$$

Fonction à plusieurs arguments les prenant séparément:

```
let f x y z = e
```

de type: $t_x \rightarrow t_y \rightarrow t_z \rightarrow t_e$

Autres équivalences

f x y est équivalent à (*f x*) *y*

```
# let moyenne x y = x+y/2;;
val moyenne : int -> int -> int = <fun>

# (moyenne 2) 4;;
- : int = 4

# moyenne 2 4;;
- : int = 4
```

Récursivité

En programmation fonctionnelle *pure*:

boucles $\Rightarrow$ écrites sous forme de *fonctions récursives*.

La fonction *f* est récursive:

si un appel à *f* apparaît dans sa propre définition.

Si *f* s'appelle elle-même, elle recommence ses calculs, donc elle boucle!

Exemples de définitions récursives

f apparaît dans sa définition(boucle récursive):

$$\texttt{let rec}\ f(a_i) = \ldots f(b_i) \ldots$$

- A est un ancêtre de B =
  si A est un parent de B, ou si A est un ancêtre d'un parent de B;
- la somme de (1..n) = n + la somme de (1..(n-1));
- il existe un chemin entre (X et Y)=
  si X=Y ou s'il y a une route directe entre X et Y, ou s'il y a une route entre X et Z et un chemin entre (Z et Y).
- X appartient au tableau T[0..N]=
  si X=T[0] ou si X appartient au tableau T[1..N]

Exemple de fonction récursive

La fonction calculant la factorielle $n!$ d'un entier naturel n:

$$\begin{aligned} 0! &= 1 \\ n! &= 1 \times 2 \times \ldots \times (n-1) \times n \end{aligned}$$

Exemple:
$4! = 1 \times 2 \times 3 \times 4 = 24.$

Sa définition récursive?

Factorielle récursive

$$n! = \begin{cases} 0 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

En Ocaml?

```
# let rec fact n =
    if n<=1 then 1  else n * fact(n-1);;
val fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

Comportément de `fact`

```
# let rec fact n =
    if n<=1 then 1  else n*fact (n-1);;
val fact : int -> int = <fun>
```

- Si $n \leq 1$, le résultat est 1 égal à 0! et 1!
- Si $n > 1$ on doit calculer $n \times fact(n-1)$
  - où $fact(n-1)$ est un *appel récursif*

On peut lire le 2ème cas par:

Pour calculer $fact(n)$ il suffit de calculer $fact(n-1)$ puis de multiplier ce résultat par n.

Déroulement d'un appel récursif

```
let rec fact n = if n<=1 then 1 else n*fact (n-1);;
```

Pour dérouler «à la main» un appel récursif:

- on remplace chaque appel par le corps de la fronction,
- où l'on remplace l'argument formel par la valeur du paramètre de l'appel.

Exemple: `fact(5)`

```
# fact(5) =  if 5<=1 then 1 else 5 * fact (5-1)
          =  5 * fact (4)
          =  5 * (if 4<=1 then 1 else 4 * fact (4-1))
          =  5 * 4 * fact(3)
....
```

Déroulement d'un appel récursif

```
let rec fact n = if n<=1 then 1 else n*fact (n-1);;
```

Déroulons un appel à `fact` avec $n = 5$:

```
# fact(5) =
    = if 5<=1 then 1 else 5 * fact (5-1)
    = 5 * fact(4)
    = 5 * (if 4<=1 then 1 else 4 * fact (4-1))
    = 5 * 4 * fact(3)
    = 5 * 4 * (if 3<=1 then 1 else 3 * fact (3-1))
    = 5 * 4 * 3 * fact (2)
    = 5 * 4 * 3 * (if 2<=1 then 1 else 2 * fact (2-1))
    = 5 * 4 * 3 * 2 * fact (1))
    = 5 * 4 * 3 * 2 * (if 1<=1 then 1 else 1 * fact (1-1))
    = 5 * 4 * 3 * 2 * 1
- : int = 120
```

Déroulement de `fact(5)`

```
# fact(5) =  5 * fact (4)
          =  5 * 4 * fact(3)
          =  5 * 4 * 3 * fact(2)
          =  5 * 4 * 3 * 2 * fact(1)
          =  5 * 4 * 3 * 2 * 1
- : int = 120
```

`fact` doit boucler tout en réalisant son calcul, puis s'arrêter:

- boucler: il y a 4 appels récursifs avec n=4,3,2 et 1.
- arrêt: le dernier appel, `fact(1)` est un cas d'arrêt, de résultat 1.
- le résultat final peut être calculé après l'arrêt:
  $5 \times 4 \times 3 \times 2 \times 1 = 120$

Caractéristiques d'une fonction récursive

Une fonction récursive a besoin de boucler, calculer un résultat, s'arrêter.

Bonnes caractérisitiques d'une fonction récursive

Elle doit possèder:

- un cas de base: sans appel récursif, où la fonction s'arrête pour retourner une solution "simple",
- un cas récursifs: où la fonction calcule son résultat par des appels récursifs.
- et à chaque appel récursif, le paramètre doit se rapprocher du cas de base (pour garantir l'arrêt).

Les bonnes propriétés de `fact`

```
let rec fact n =
   if n<=1 then 1        Cas de base
   else n*fact (n-1) Cas récursif
```

- cas de base: si $n \leq 1$, on connaît la solution $\Rightarrow 1$;
- cas récursifs: $n > 1$, solution qui combine le paramètre courant n et le résultat de l'appel récursif $fact(n-1)$;
- le paramètre décroit: à chaque appel récursif, le paramètre utilisé est $n-1$ qui se rapproche ainsi du cas de base.

Comment écrire une fonction récursive?

Les questions à élucider:

1. Quel est le (ou les) paramètre *p* de la fonction?
   Ce sera aussi celui de la récursion;
2. Quelle est la valeur de base de ce paramètre pour laquelle nous pouvons donner la solution sans boucler?
   Elle servira à établir le cas de base;
3. Comment exprimmer la solution du cas récursif? Il faut combiner le paramètre courant *p* avec le résultat d'un appel récursif;
4. Quelle valeur donner au paramètre de l'appel récursif pour qu'il se rapproche du cas de base?

L'exemple de `fact`

1. Le paramètre $p = n$. Pour calculer la factorielle de n, le paramètre doit être n;
2. La valeur de base de ce paramètre: $n = 0$ et $n = 1$, dont le résultat à renvoyer est 1;
3. Comment exprimmer la solution du cas récursif?
   $\Rightarrow n \times fact(n-1)$
4. Quelle valeur donner au paramètre de l'appel récursif pour qu'il se rapproche du cas de base?
   $\Rightarrow (n-1)$

Schémas récursifs les plus courants

Beaucoup de fonctions récursives suivent un petit nombre de schémas classiques.

Schéma 1: Fonction récursive sur *n* (entier)

- La valeur de base est souvent n=0 ou n=1;
- Si ($n > base$), la solution s'obtient en combinant (par addition, multiplication, comparaison, etc) une valeur qui dépend du paramètre courant *n* avec le résultat d'un appel récursif.
- le paramètre de l'appel récursif est souvent ($n - 1$).

Fonctions récursives sur n

Problème: Calculer la somme des entiers de 0 à n:

- Le paramètre de la fonction est n;
- La valeur de base est n=0, qui donne en résultat 0;
- Si $(n > 0)$, la solution s'obtient par addition du n courant et de la somme jusqu'à $(n-1)$.
- le paramètre de l'appel récursif se rapproche de la base car il est $(n-1)$.

$$somme(n) = \begin{cases} 0 & \text{si } n = 0, \\ n + somme(n-1) & \text{si } n > 0 \end{cases}$$

Fonctions récursives sur *n*

Problème: Calculer la somme des entiers de 0 à n:

$$somme(n) = \begin{cases} 0 & \text{si } n = 0, \\ n + somme(n-1) & \text{si } n > 0 \end{cases}$$

En Ocaml?

```
# let rec somme n =
     if n<=0 then 0  else n + somme (n-1);;
val somme : int -> int = <fun>

# somme 6;;
- : int = 21
```

Fonctions sur structure composite de taille fixe

Problème

Tester si un caractère *c* apparaît dans une chaîne *s*.

- `s.[i]`: caractère de position *i* de la chaîne *s*;
- 1er caractère à 0;
- `String.length s`: longueur de la chaîne *s*.

Quels sont: le paramètre récursif?,
la valeur de base,
la solution du cas récursif?

Fonctions sur tableaux et chaînes

Problème

Tester si un caractère c apparaît dans une chaîne s.

Une chaîne est une structure composite de taille fixe
$\Rightarrow$ parcours sur $n = taille(s)$

- Méthode: parcours sur indices taille(s)-1 jusqu'à 0.
- si c n'est pas à l'indice courant, passer au précédant;
- paramètre récursif: entier naturel n,

$\Rightarrow$ c'est donc une fonction récursive sur n (schéma 1).

Fonctions sur tableaux et chaînes

Méthode: parcours de la chaîne du dernier au premier caractère.

- paramètre n: indice de début du parcours;
- autres paramètres: caractère c et chaîne s;
- Base: si on est à la position avant le 1er caractère: $n < 0$, on a fini;
- Récursion: si $n \geq 0$, soit le caractère se trouve à $s[n]$, soit dans la sous-chaîne qui va de $0..(n-1)$.

$$dans(c, s, n) = \begin{cases} false & \text{si } n < 0, \\ \texttt{s[n] = c ||} \ dans(c, s, n-1) & \text{si } n \geq 0 \end{cases}$$

Ici, le cas $n < 0$ signifie que s est la chaîne vide.

Fonction `dans`

Tester si un caractère c apparaît dans une chaîne s.

$$dans(c, s, n) = \begin{cases} false & \text{si } n < 0, \\ \texttt{s.[n] = c ||} \ dans(c, s, n-1) & \text{si } n \geq 0 \end{cases}$$

En Ocaml?

```
# let rec dans (c,s,n) =
    if n<0 then false
    else s.[n]=c || dans(c,s, n-1);;
val dans : char * string * int -> bool = <fun>

# dans('u', "salut", 4);;
- : bool = true
```

Fonction `dans`: non satisfaisante

- on doit lui passer l'indice *n* pour démarrer la recherche,
- si mauvais indice (trop petit ou trop grand) $\Rightarrow$ erreur ou réponse incorrecte;
- la fonction devrait prendre uniquement la chaîne et le caractère.

```
# let rec dans (c,s,n) =
   if n<0 then false else s.[n]=c || dans(c,s, n-1);;

# dans('u', "salut", 4);;
- : bool = true

# dans('u', "salut", 1);;
- : bool = false

# dans('u', "salut", 5);;
Exception: Invalid_argument "index out of bounds".
```

Solution: Fonctions récursives locales

- définir une fonction `dansChaine` (non récursive) prenant en paramère *c* et *s* uniquement;
- qui définit `dans` localement;
- et qui l'invoque avec la bonne valeur de *n* = taille(s)-1
- l'itération récursive est réalisée par `dans`, la fonction globale ne fait que l'encapsuler et l'appeller avec les bons arguments.

```
# let dansChaine (c,s) =
   let rec dans n =
     if n<0 then false
     else s.[n]=c || dans(n-1)
   in dans(String.length s -1);;

# dansChaine('u', "salut");;
- : bool = true
```

Fonctions récursives locales: quand?

La fonction *dans*(c, s, n) a 3 paramètres:

- c et s sont les seuls paramètres que l'on souhaite avoir à donner; ils ne changent pas pendant la récursion;
- n est un paramètre récursif (change pendant la récursion);
- on peut séparer (itération récursive sur n) et (fonction sur c, s).

Schéma 2: Récursivité locale

- Si plusieurs paramètres dont certains inchangés dans la récursion,
- $\Rightarrow$ on définit une fonction récursive locale uniquement sur les paramètres récursifs.

Fonctions sur tableaux avec résultat composite

Problème:

1. Parcours d'une structure composite de taille fixe (tableau, chaîne);
2. extraire certains éléments,
3. fabriquer avec $\Rightarrow$ nouvelle structure composite (résultat).

Extraire les chiffres d'une chaîne s

Méthode: parcours récursif sur n en collectant les caractères chiffres en les concatenant au résultat.

- $n = taille(s)$: paramètre récursif (schéma 1)
- s: paramètre «externe»
- définir une fonction locale de parcours récursif (schéma 2)
- résultat: nouvelle chaîne obtenue par concaténation des chiffres trouvés;

Fonctions auxiliares

```
let digit c = '0' <= c && c <= '9';;

let str c = String.make 1 c;;
```

- `digit`: teste si un caractère est un chiffre;
- `str`: transforme un caractère en chaîne;
- `p(s,i)`: parcourt la chaîne (de i à 0) en concaténant tous ses caractères chiffres dans son résultat.

Fonction de parcours

- si $i < 0 \Rightarrow$ chaîne vide (on a fini);
- sinon
  - si `(s.[i])` est un chiffre $\Rightarrow$ on le concatène au résultat d'un appel récursif sur $(i - 1)$;
  - sinon $\Rightarrow$ appel récursif sur $(i - 1)$;

$$pi = \begin{cases} \texttt{""} & \text{si } i < 0 \\ \texttt{p (i-1)} & \text{si } i > 0 \wedge \neg \texttt{digit(s.[i])} \\ \texttt{p(i-1)\^{}str(s.[i])} & \text{si } i > 0 \wedge \texttt{digit(s.[i])} \end{cases}$$

Fonction de parcours

$$p(i) = \begin{cases} \texttt{""} & \text{si } i < 0 \\ \texttt{p(i-1)} & \text{si } i > 0 \wedge \neg \texttt{digit(s.[i])} \\ \texttt{p(i-1)\^{}str(s.[i])} & \text{si } i > 0 \wedge \texttt{digit(s.[i])} \end{cases}$$

En Ocaml:

```
let rec p i =
  if i<0 then ""
  else let ce = p(i-1) in
     if digit(s.[i]) then ce^(str s.[i])
     else ce
```

Fonction `extraitChiffres`

```
let rec extraitChiffres s =
   let digit c = '0' <= c && c <= '9'in
   let str c = String.make 1 c in
   let rec p i =
     if i<0 then ""
     else let ce = p(i-1) in
        if digit(s.[i]) then ce^(str s.[i])
        else ce
   in p (String.length s -1);;
val extraitChiffres : string -> string = <fun>

# extraitChiffres "salut";;
- : string = ""

# extraitChiffres "ab 15 cde 5 code4";;
- : string = "1554"
```

Exemple

```
extraitChiffres "a4b7"
 = p(String.length "a4b2" -1)
 = p(3)
 = if digit(s.[3]) then p(2)^str(s[3]) else ...
 = if digit('7') then p(2)^str('7') else ...
 = p(2)^str('7')
 = (if digit(s.[2]) ..  else p(1))^str('7')
 = (if digit('b') ..  else p(1))^str('7')
 = p(1)^str('7')
 = (if digit(s.[1]) then p(0)^str(s.[1])..)^str('7')
 = (if digit('4') then p(0)^str('4') ..)^str('7')
 = (p(0)^str('4'))^str('7')
 = (if digit('a') ..  else p(-1))^('4')^str('7')
 = (p(-1))^str('4')^str('7')
 = (if (-1)<0 then "" ..)^str('4')^str('7')
 = ("")^str('4')^str('7')
 = "" ^  "4"     ^"7"
- : string = "47"
```

Les palindromes

Un palindrome est un est un texte dont la succession des lettres est la même qu'il soit lu de gauche à droite ou de droite à gauche.

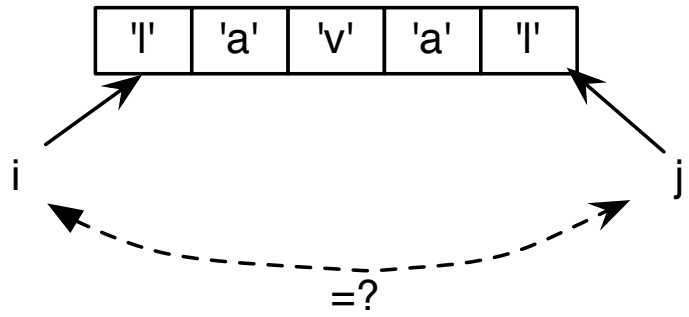
Exemples

- "abba"
- "laval"
- "eluparcettecrapule"

Problème: Tester si une chaîne de caractères est un palindrome.

Tester si une chaîne est un palindrome

Algorithme: deux indices i, j initialisées avec premier et dernier indices de la chaîne.



- tant que $i < j$ et les caractères à ces indices sont égaux, incrémenter i, décrémenter j;
- si les caractères sont différents, arrêter avec `false`;
- si $i \geq j$, on a comparé tous les caractères sans tomber sur des caractères différents: on termine avec `true`.

Paramètres et fonctions auxiliaires

- i, j: paramètres récursifs (les seuls à changer à chaque itération),
- ils déterminent la condition d'arrêt.

Nous utilisons une fonction récursive locale `palinRec` qui prend i, j en paramètres:

- elle testera les valeurs de i, j,
- si l'on peut continuer, nouvel appel récursif avec les nouveaux paramètres $i + 1$ et $j - 1$.

Fonction de parcours

- elle testera les valeurs de i, j,
- si l'on peut continuer, nouvel appel récursif avec les nouveaux paramètres $i + 1$ et $j - 1$.

$$p(i,j) = \begin{cases} \texttt{true} & \text{si } i \geq j \\ \texttt{s.[i]=s.[j]} \ \wedge \ p(i+1, j-1) & \text{si } i < j \end{cases}$$

Fonction `palindrome`

La fonction `palindrome`

- prend *s* en paramètre,
- et définit localement `p` sur *i*, *j*.

```
let palindrome s =
  let rec p (i, j) =
     if (i<j) then
        if (s.[i] = s.[j])
            then p ((i+1),(j-1))
        else false
     else true
  in p (0, (String.length s - 1));;
val palindrome : string -> bool = <fun>
```

Version plus compacte de `palindrome`

Nous utilisons les opérateurs logiques "paresseux".

```
let palindrome s =
  let rec palinRec (i, j) =
     i>=j || (s.[i] = s.[j] && palinRec ((i+1),(j-1)))
  in palinRec (0, (String.length s - 1));;
val palindrome : string -> bool = <fun>

# palindrome "aba";;
- : bool = true
```

Exemples

```
# palindrome "abc";;
- : bool = false

# palindrome "aba";;
- : bool = true

# palindrome "abba";;
- : bool = true
```

Exemple: chaîne de taille impair

```
let palindrome s =
  let rec p (i, j) =
     i>=j || (s.[i] = s.[j] && p ((i+1),(j-1)))
  in p (0, (String.length s - 1))
```

```
palindrome ”aba”
```

```
palindrome "aba" =
  =  p (0, (String.length "aba" - 1))
  =  p (0,2)
  =  0>=2  || (s.[0] = s.[2] && p ((0+1),(2-1)))
  =  false || ('a' = 'a' && p (1,1)
  =  ('a' = 'a' && p (1,1)
  =  true       && ( 1>=1 || ..)
  =  ( 1>=1 || ..)
  =  ( true || ..)
  =  true
```

Exemple: chaîne de taille pair

```
let palindrome s =
  let rec p (i, j) =
     i>=j || (s.[i] = s.[j] && p ((i+1),(j-1)))
  in p (0, (String.length s - 1))
```

```
palindrome "abba"
```

```
palindrome "aba" =
  =  p (0, (String.length "abba" - 1))
  =  p (0,3)
  =  0>=3  || (s.[0] = s.[3] && p ((0+1),(3-1)))
  =  false || 'a' = 'a' && p ((1),(2)))
  =  ('a' = 'a' && p (1,2)
  =       true   && ( 1>=2 ||(s.[1] = s.[2] && p(1+1,2-1))
  =  ( 1>=2 ||('b' = 'b' && p(2,1))
  =  (false ||('b' = 'b' && p(2,1)))
  =  ('b' = 'b' && p(2,1))
  =  p(2,1)
  =  (2>=1 || ...))
  =  true
```

Correction de fonctions récursives

Un programme est *correcte* s'il calcule ce que nous voulons qu'il calcule.

- Pour nous coinvaincre de la correction d'une fonction récursive nous devons raisonner sur les calculs de la fonction pour *tous les cas d'entrée possibles*.
- Nous montrons un exemple de *raisonnement par récurrence*

Raisonnement par récurrence

Nous procédons en deux temps:

- montrer que la fonction se comporte correctement dans son *cas le plus simple*;
- monter que *si la fonction est correcte pour un cas donné*, alors elle aussi correcte pour le *cas suivant*.

Raisonnement par récurrence

Pour la fonction `fact` cela donne:

- **Cas simples**: `fact(0), fact(1)` où $n < 2$. La réponse est 1=0! = 1!. Donc `fact` est correcte pour $n = 0$ et $n = 1$
- **Cas avec hypothèse**: Supposons que `fact(n-1)` calcule $(n-1)!$ Nous devons monter que le cas suivant `fact(n)` = $n!$
  - Comme $n \geq 2$ la réponse calculée est `fact(n) = n*fact(n-1)`.
  - Comme nous avons supposé que `fact(n-1)` = $(n-1)!$
  - nous obtenons que la réponse calculée est `fact(n) = n*fact(n-1)` = $n * (n-1)! = n!$

Inférence de types

- En Ocaml, il n'est pas nécessaire de déclarer les types des identificateurs, ou paramètres,
- le typeur infère leur type d'après leur contexte.

Exemple d'inférence de types

```
let f(x) = x + 1
```

Sachant que `+` est défini uniquement sur les entiers, et a le type `+`: $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$, le typeur déduit les contraintes:

- le type de `f` est de la forme:

$$\texttt{f} : t_x \rightarrow t_{corps}$$

  où $t_x \approx$ type de x (argument), et $t_{corps} \approx$ type du corps.
- de `x+1` on obtient que `x` doit être de type `int` $\Rightarrow \boxed{t_x = \texttt{int}}$,
- comme `x+1` est le le corps, on obtient que celui-ci est de type `int` $\Rightarrow \boxed{t_{corps} = \texttt{int}}$.

Conclusion: $t_x = \texttt{int}$ et $t_{corps} = \texttt{int}$, donc

$$\texttt{f} : \texttt{int} \rightarrow \texttt{int}$$

□

Généricité

- *ne pas restreindre inutilement* le type des données et fonctions,
- au contraire: les rendre *génériques à toute valeur compatible* avec leur définition.
- on utilise des *paramètres de type*:
  - inconnus à la définition (donc génériques),
  - précisés (*instanciés*) lors d'un appel ou utilisation concrète.

Généricité $\approx$ *Polymorphisme paramétrique*:

- un type est *polymorphe* s'il peut avoir *plusieurs formes*;
- *paramétrique*: utilisation de paramètres de type.

Généricité en programmation

Mécanisme de typage présent en C++ (*templates*), Ada (*generics*), Java 1.5.

Java 1.5: Type d'une cellule: `class Cellule<A>`.
Le *type de son contenu* dépend du paramètre de type A

```
class Cellule<A> {A contenu; Cellule suivant};
```

C++: Type de la fonction `identity`: $A \rightarrow A$

```
template <typename A>
A identity (A x) {
  return x;
}
```

Polymorphisme paramétrique en Ocaml

- Implanté de *manière automatique* par le typeur,
- le type inféré pour toute phrase est *le plus général possible* compatible avec la définition qui est typée.

```
let premier (x,y) = x
```

doit être applicable sur *n'importe quel type des paires*:

$$\texttt{premier(1, true)} \Rightarrow \texttt{1}$$

$$\texttt{premier("abc", 's')} \Rightarrow \texttt{"abc"}$$

Son type en Ocaml:

```
# let premier (x,y) = x;;
val premier: 'a * 'b -> 'a
```

Paramètres de type en Ocaml

```
# let premier (x,y) = x;;
val premier: 'a * 'b -> 'a

premier (1, true);;
-: int = 1

premier("abc", 's');;
-: string = "abc"
```

- `'a` et `'b` sont des *paramètres de type*,
- introduits par le typeur en l'absence des contraintes trouvées lors du typage;
- ils seront instanciés par des types spécifiques lors de chaque utilisation;
- et à chaque utilisation l'instantiation peut être différente.

Instantiation des paramètres de type

Instantiation de paramètres: leur donner des "valeurs" spécifiques.

- En C++, Ada, Java: c'est fait par le programmeur;
- En Ocaml, c'est *inféré* lors de chaque utilisation;
- et à chaque utilisation l'instantiation peut être différente.

```
premier (1, true):   'a ↦ int      'b ↦ bool
premier ("ab", 0):   'a ↦ string   'b ↦ int
```

Mais ceci est-il bien typée?

```
let x = premier (1, true) + premier(2,"ab");;

let y = premier (1, true) + premier("ab",2);;
```

Précision du polymorphisme paramétrique

Ceci est-il bien typée? Comment les variables de type sont-elles instanciées?

```
# let x = premier (1, true) + premier(2,"ab");;
val x : int = 3

# let y = premier (1, true) + premier("ab",2);;
                              ----------------
Error: This expression has type string but an
expression was expected of type    int
```

Conclusion: un type polymorphe paramétrique n'est pas un type laxiste mais donne un typage précis.

Apparté: existe-t-il d'autres sortes de polymorphisme?

Les fonctions en paramètre

Fonctions ≡valeurs de première classe

elles peuvent être passés en argument à d'autres fonctions.

Exemple: Determiner à partir de 2 notes si un élève est reçu + sa note finale. Retourner une chaîne contenant ces deux résultats.

Les paramètres nécessaires:

- `(n1,n2)`: une paire de notes;
- `modeCalcul`: fonction à appliquer sur la paire des notes afin de calculer la note finale;
- `min`: note minimale pour être reçu;

Les paramètres fonctionnels

- `(n1,n2)`: une paire de notes;
- `modeCalcul`: fonction à appliquer sur la paire des notes afin de calculer la note finale.
  Exemples:
  - la fonction qui calcule la moyenne de deux notes;
  - la fonction qui calcule une moyenne pondérée avec 30% pour la 1ère note et 70% pour la 2ème;
- `min, max`: notes minimale pour être reçu et maximale possible.
  Exemple: min = 10 sur max =20 ou 45 sur 100.

`modeCalcul` est un paramètre fonctionnel

La fonction `resEleve`

```
# let resEleve modeCalcul (min,max) (n1,n2)=
  let nfinale = modeCalcul (n1,n2) in
  let recu = if (nfinale >= min) then "Recu"
             else "Non recu" in
  let note = string_of_float(nfinale) in
  let m =  string_of_float(max)
  in recu^" avec "^note^"/"^m;;
```

- `nfinale:` résultat d'appliquer le mode de calcul (fonction `modeCalcul`) sur la paire des notes;
- `recu:` contient "recu" ou "non recu"
- `note:` note finale transformée en `string`;
- `m`: note maximale transformée en `string`;
- ligne finale: c'est le résultat renvoyé par la fonction.

Calculer une moyenne simple

```
let resEleve modeCalcul (min,max) (n1,n2) = ...
```

On déclare le mode de calcul correspondant à une moyenne simple:

```
# let moySimple (x,y) = (x+.y)/.2.0;;
```

On l'utilise pour calculer si un élève est reçu en anglais:

```
#let noteAnglais=
    resEleve moySimple(10.0,20.0) (11.0,12.0);;
val noteAnglais : string = "Recu avec 11.5/20."
```

Les paramètres dans cet appel:

- modeCalcul = `moySimple`
- les notes mimales et maximale possibles: (min = 10.0; max = 20.0)
- les notes de l'élève : (n1 = 11.0, n2= 12.0)

Calculer une moyenne ponderée

```
let resEleve modeCalcul (min,max) (n1,n2) = ...
```

On déclare le mode de calcul correspondant à une moyenne ponderée:

```
# let moyPonderee (x,y) = (x*.0.4+.y*.0.6);;
```

On l'utilise pour calculer si un élève est reçu en maths:

```
# let noteMath =
    resEleve moyPonderee (45.0,100.0) (65.0, 42.0);;
val noteMath : string = "Recu avec 51.2/100."
```

Les paramètres dans cet appel:

- modeCalcul = `moyPonderee` (0.4 pour la 1ère note, 0.6 pour la 2ème);
- les notes mimales et maximale possibles: (min = 45.0; max = 100.0)
- les notes de l'élève : (n1 = 65.0, n2= 42.0)

Type de `resEleve`

```
# let resEleve modeCalcul (min,max) (n1,n2)=
    let nfinale = modeCalcul (n1,n2) in ...
```

```
resEleve : ('a*'b → float) → float*float → 'a*'b → string
```

- `modeCalcul` (type d'une fonction): `('a*'b → float)`
- (`min, max`): `float*float`
- `(n1,n2)`: type d'une paire: `'a*'b`
- résultat de la fonction: `string`

Le type de l'argument fonctionnel est mis entre parenthèses.

Fonctions paramétrées par des fonctions

- Elles sont très générales;
- On peut donc les réutiliser dans différents contextes;
- Le programme gagne en concision, clarté et est plus facile à maintenir et à débogguer.

Fonctions partielles (en mathématiques)

fonction partielle (maths)

Une fonction est partielle si elle n'est pas définie pour toutes les valeurs possibles de ses paramètres.

Exemples:

- la fonction qui calcule l'indice d'un caractère *c* dans une chaîne *s*: elle n'est pas définie si *c* n'apparaît pas dans *s*;
- la fonction qui extrait le premier caractèere d'une chaîne *s*: n'est pas définie si *s* est vide;

Une fonction qui nést pas partielle est totale.

Fonctions partielles en Ocaml

Fonction partielle (programmation)

- incapable de rendre un résultat pour certains arguments car non définie;
- exemple: impossible de renvoyer le 1er caractère d'une chaîne si elle est vide;
- la fonction doit donc s'interrompre en levant une exception;

En Ocaml:
(`failwith message`), lève l'exception `Failure` qu'on fait suivre d'une chaîne `message`. Cela arrête l' exécution de la fonction.

Fonctions partielles en Ocaml (exemple)

- la fonction $n!$ est définie uniquement si $n \geq 0$;
- on fera donc échouer la fonction Ocaml correspondante pour les cas où $n < 0$

```
let rec fact n =
   if n<0 then failwith "fact"
   else if n<=1 then 1
   else n*fact(n-1)

# fact 0;;
- : int = 1

# fact (-2);;
Exception: Failure "fact".
```

Cas d'échec $\neq$ cas de base

- partielle: fonction renvoie l'indice du caractère *c* dans la chaîne *s*
- et celle qui teste si un caractère *c* est dans *s*? partielle ou totale?

  si le caractère est absent elle renvoie un résultat (=false)

$\Rightarrow$ elle n'est donc pas partielle, elle est totale.

```
# let dansChaine (c,s) =
   let rec dans n =
     if n<0 then false
     else s.[n]=c || dans(n-1)
   in dans(String.length s -1);;

# dansChaine('k', "salut");;
- : bool = false
```

Cas d'échec $\neq$ cas de base

Comment se comporte cette version?

```
# let dansChaineBis (c,s) =
   let rec dansBis n =
     if n<0 then failwith "dansBis"
     else s.[n]=c || dansBis(n-1)
   in dansBis(String.length s -1);;

# dansChaineBis('u', "salut");;
- : bool = true

# dansChaineBis('k', "salut");;
Exception: Failure "dansBis".
```

Pourquoi?

Le cas de base ($n < 0$) qui doit rendre un vrai résultat (false) a été transformé en cas d'échec.

Type enregistrement

Agrégats de valeurs hétérogènes, chacune nommée et accessible par un nom de champ ou étiquette.

Déclaration d'un type enregistrement: en spécifiant ses noms de champs et leurs types.

```
# type client={numero:int; nom: string; solde: float};;
type client = { numero: int; nom: string; solde: float }
```

Valeurs enregistrement

Construction de valeurs: On doit donner des valeurs pour chacun de ses champs déclarés:

```
#let durand = { numero = 265; nom = "Durand";
                solde=  0.0};;
val durand : client={numero=265; nom="Durand"; solde=0}
```

Accès aux champs: Avec la notation `e.champ`

```
# durand.numero;;
- : int = 265
```

Enregistrements non modifiables

à moins de préciser le contraire, les champs d'enregistrement sont *constants*.

On ne peut pas changer leurs valeurs, mais on peut les recopier:

```
# let nouveauSolde c s =
      {numero=c.numero; nom = c.nom; solde = s};;
 val nouveauSolde: client -> float -> client

# let d={numero=265;nom="Durand";solde= 0.0};;
val d : client={numero=265; nom="Durand"; solde=0}

# let dBis = nouveauSolde d 50.34;;
val dBis: client={numero=265;nom="Durand";solde=50.34}
```

Structures des données polymorphes

Déclaration d'un type polymorphe en Ocaml

- on *déclare* le param. de type `'a` *devant le nom du type*;
- `'a` doit apparaître dans le corps de la déclaration.

Exemple: cellules avec deux champs de type *A*:

- Java 1.5: on déclare `class Cellule<A>`.

```
class Cellule<A> {A contenu; A moisPrec};
```

- Ocaml: on déclare `'a cellule`

```
type 'a cellule = {contenu: 'a; moisPrec: 'a};
```

Structures des données polymorphes

```
# type 'a cellule = {cont: 'a; moisPre: 'a};;

# let c1 = {cont = 3; moisPrec = 7};;
val c1: int cellule = {cont = 3; moisPre= 7}

# let c2 = {cont= "Paris"; moisPre= "Lyon"};;
val c2: string cellule = {cont= "Paris"; moisPre= "Lyon"}

# let c3 = {cont=3; moisPre= "Lyon"};;
Error: This expression has type string but
an expression was expected of type  int
```

Le typeur infère l'instantiation pour chaque utilisation:

- `'a` $\mapsto$ `int` $\Rightarrow$ `c1:int cellule`,
- `'a` $\mapsto$ `string` $\Rightarrow$ `c2:string cellule`
- `'a` $\mapsto$ `int` et `'a` $\mapsto$ `string` $\Rightarrow$ ERREUR DE TYPAGE

Quelques fonctions polymorphes prédéfinies

```
#fst;;
- : 'a * 'b -> 'a = <fun>

#snd;;
- : 'a * 'b -> 'b = <fun>

# (=);;
- : 'a -> 'a -> bool = <fun>

# (>);;
- : 'a -> 'a -> bool = <fun>
```

Le filtrage

- Mécanisme de reconnaissance syntaxique:
  - appliqué sur une donnée structurée;
  - afin de reocnnaître un cas syntaxique particulier de cette donnée;
  - et d'en extraire certaines composantes.
- Utilise la notion de patron ou motif de reconnaissance syntaxique,
- On recherche la correspondance entre une valeur et un motif.

Exemple

- (10,"Mars") : donée structurée;
- (j ,"Mars") : motif syntaxique reconnaîssant toute paire dont la 2ème composante est "Mars"

Motifs ou patrons syntaxiques

Motif

- Patron décrivant une forme syntaxique possible d'une donnée;
- pouvant contenir des constantes et des variables;
- les variables seront liés à des morceaux de la donnée comparée.

Exemple: Motif **(1,m)**

- contient: constante 1, variable m
- décrit: *toutes les paires avec 1 en 1ère composante*;
- les constantes devront être présentes sur la donnée,
- les variables seront liées aux parties de la donnée en dehors de ces contanstes.

Exemples de motifs

Motif	il permet de reconnaître
`2`	constante 2
`(0,x)`	toute paire avec un 0 en 1ère composante
`y`	toute valeur
`{cle=k; contenu=c}`	enregistrement avec champs `cle` et `contenu`
`{cle=7}`	enregistrement avec (au moins) un champ `cle` égal à 7
`[0; _ ]`	liste avec 2 éléments dont le 1er est 0

Filtrage: reconnaissance + liaison

Filtrage = reconnaissance + liaison

- **Recherche de correspondance** entre une donnée et un motif;
- le filtrage réussit s'il y a correspondance;
- si le motif contient des variables, elles seront liés aux composantes de même position dans la donnée,
- et pourront être employées pour calculer la valeur rendue par l'expression de filtrage.

Une donnée reconnue est dite est *filtrée* (*matched*) par le motif.

Exemple: la donnée (0, 4)

- est filtrée par le motif (0,x);
- et la variable x du motif devient liée à 4

Filtrage: exemples

Motif	Valeur comparée	Réussite	Liaisons
1	1	oui	aucune
x	1	oui	x=1
`(0,x,y)`	`(0,1,6)`	oui	x=1, y=6
`{cle=k; cont=c}`	`{cle=5; cont="h"}`	oui	k=5, c="h"
0	1	échec	
`(1,x)`	`(1,2,3)`	échec	
`(_,x,_)`	`(0,1,6)`	oui	x=1

Notation : si *val* est filtré par *mo*, nous noterons ceci par $val \triangleright mo \Rightarrow OK$, et dans le cas contraire par $val \triangleright mo \Rightarrow Echec$

Exemple:
$(0, 1, 6) \triangleright (0, x, y) \Rightarrow OK$ avec $x = 1$ et $y = 6$.

L'expression `match with`

```
match expr
 with motif_1 -> a_1
 |    motif_2 -> a_2
 |    ....
 |    _        -> a_autres
```

- Compare `expr` à chaque motif (en respectant l'ordre);
- si `motif`$_i$ est le premier à filtrer expr, le résultat de l'expression est a_i;
- sinon, s'il n'y a pas de motif qui filtre expr,
- soit il y a un cas qui filtre tout _, et le résultat est a_{autres},
- soit, le résultat de toute l'expression est un échec.

Expression `match with`: exemple

Calculons le lendemain de la date représentée par (31,3,2010):
(on suppose donnée la fonction `joursMois`)

```
match (31,3,2010)
 with (31,12,a) -> (1,1,a+1)
  |    (31, m, a)-> (1, m+1, a)
  |    (j,m,a)    -> if (j< joursMois m a)
                       then (j+1,m,a)
                       else (1,m+1,a)
- : int * int * int = (1, 4, 2010)
```

1. Motif 1: $(31, 3, 2010) \triangleright (31, 12, a) \Rightarrow \mathit{Echec}$
2. Motif 2: $(31, 3, 2010) \triangleright (31, m, a) \Rightarrow \mathit{OK}$ avec $m = 3$, $a = 2010$.
3. le résultat calculé est donc $(1, m + 1, a) = (1, 4, 2010)$

Programmer par cas avec le filtrage

Fonctions par filtrage

- But: programmer *par cas*.
- La fonction est définie par une *expression de filtrage sur son paramètre*;
- son corps est donc une expression `match with`
- si plusieurs paramètres, le filtrage se fait sur tous au même temps (p.e. dans un n-uplet);

```
let f x =
 match x
  with motif_1 -> res_1
  |    motif_2 -> res_2
  |    ....
  |    _        -> res_tous_autres_cas
```

Exemple de fonction par filtrage

Cette fonction filtre son argument (une paire) afin de réaliser l'addition de ses composantes:

```
# let somme x y =  match (x,y)
  with (0,n) -> n
  |    (n,0) -> n
  |    (a,b) -> a+b;;
val somme : int -> int -> int = < fun >

# somme 3 0;;
- : int = 3
```

Comportement du filtrage

```
# let somme x y =  match (x,y)
  with (0,n) -> n
  |    (n,0) -> n
  |    (a,b) -> a+b;;
```

Comportement de l'appel `(somme 3 0)` $\Rightarrow$

- on exécute `(match(3,0) with ...)` $\Rightarrow$
- qui compare `(3,0)` avec chacun des motifs:
  1. motif 1: `(0,n)` comparé à `(3,0)` $\Rightarrow$ échec,
  2. motif 2: `(n,0)` comparé à `(3,0)` $\Rightarrow$ réussit avec $\boxed{\text{n=3}}$
     $\Rightarrow$ on exécute la partie droite de ce motif
     $\Rightarrow$ le résultat renvoyé est $\boxed{3}$

Date du lendemain par filtrage

Prends une date (triplet) et calcule (nouveau triplet) la date du lendemain.

```
# let lendemain (j,m,a) = match (j,m,a)
  with (31,12,a) -> (1,1,a+1)
  |    (31, m, a)-> (1, m+1, a)
  |    (j,m,a)   -> if (j< joursMois m a)
                      then (j+1,m,  a)
                      else (1  ,m+1,a)

# lendemain (3,2,2010);;
- : int * int * int = (4, 2, 2010)

# lendemain (28,2,2010);;
- : int * int * int = (1, 3, 2010)

# lendemain (28,2,2004);;
- : int * int * int = (29, 2, 2004)
```

La fonction `JoursMois m a` calcule le nombre total de jours du mois m de l'année a.

Filtrage incomplet

Un filtrage est *incomplet* si tous les cas possibles d'une valeur ne sont pas répertories.

Un avertissement est indiqué à la compilation.

```
let est_null x =
  match x
  with  0 -> true
   |    1 -> false
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
val est_null : int -> bool = <fun>
```

Filtrage incomplet(suite)

Un filtrage incomplet produira une erreur à l'exécution en cas d'appel sur une expression non prévue par les motifs:

```
# est_null 1;;
- : bool = false

# est_null 3;;
Exception: Match_failure ("", 2, 2).
```

Pour compléter ce filtrage:

```
let est_null x =
  match x
  with  0 -> true
   |    _ -> false
val est_null : int -> bool = <fun>
```

Exemple: filtrage sur enregistrements

Fonctions pour tester si un client est débiteur ou créditeur.

```
type client = {numero :int; nom:string; solde:float; }

# let crediteur {numero=_;nom= _; solde=s}= s>0.0;;
val crediteur : client -> bool = <fun>

# let debiteur {solde = s} = s < 0.0;;
val debiteur : client -> bool = <fun>
```

- les fonctions `crediteur` et `debiteur` filtrent leur argument directement dans leur entête;
- ici, le filtrage ne correspond pas à une reconnaissance de cas syntaxique;
- il est plutôt utilisé pour lier des composantes du paramètre;

Un type prédéfini polymorphe: les listes

Les listes

Suites d'éléments de même type, accéssibles séquentiellement.

C'est un type des données polymorphe:

`'a list` $\equiv$ *liste d'éléments de type* `'a`

On peut spécialiser le paramètre de type `'a` et obtenir des listes:

- d'entiers: `int list`,
- de paires: `bool*int list`,
- de dates: `date list`, etc...

Les listes: syntaxe

Constantes:

- `[]` est la liste vide.
- Deux notations pour les listes non vides:
  - `[a; b; c]` notation courante pour les constantes.
  - `a::l` notation pour le filtrage.

Opérateurs: `@` de concaténation de deux listes.

Fonctions prédéfinies: dans le module `List`.

Exemples

Une liste d'entiers:

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

Concaténation de deux listes:

```
# ["a";"bc"] @ ["bonjour"];;
- : string list = ["a"; "bc"; "bonjour"]
```

La liste vide est polymorphe:

```
# [];;
- : 'a list = []

# [1]@[];;
- : int list = [1]
```

Exemples (suite)

La liste vide est polymorphe:

```
# ["ab";"cd"]@[];;
- : string list = ["ab"; "cd"]
```

Erreurs:

```
# [(1,2);3];;
This expression has type int but is here used with type int

# [2]@['a'];;
This expression has type char but is here used with type in

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

Exemples(suite)

Appel de la fonction `length` du module `List`:

```
# List.length [1;2;3];;
- : int = 3
```

Liste de paires:

```
# [(1,'a'); (23, 'c')];;
- : (int * char) list = [(1, 'a'); (23, 'c')]
```

Liste de listes:

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

Construire une liste par la notation `a::l`

`::` est le *constructeur* des listes.

La liste a ::l

est obtenue à partir de:

- un élément *a*, qui sera le premier de la liste construite,
- une liste *l*, qui sera la suite de la liste construite.

```
# 1::[];;
- : int list = [1]

# 1::[3;5];;
- : int list = [1; 3; 5]

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

Fonctions sur les listes

fonctions sur une liste

- Très souvent, définie par filtrage,
- avec au moins deux cas:
  - la liste vide $\Rightarrow$ correspond au *motif* `[]`
  - la liste non vide (au moins un élément) $\Rightarrow$ correspond au motif `a::l`, où a est son premier élément, et l est le reste.

Exemple: fonction `premier`

La fonction qui extrait le premier élément d'une liste, compare son argument avec les deux cas possibles:

- Si elle est vide, la fonction échoue.
- Si elle n'est pas vide, elle est nécessairement de la forme `e:: reste`, où `e` est le premier élément de la liste.

Exemple: fonction `premier`

```
# let premier l =
match l
with []         -> failwith "premier"
   |  e::reste -> e;;
val premier : 'a list -> 'a = <fun>
```

(1) Appel `(premier [3;4;5])` $\Rightarrow$

1. `[3;4;5]` $\triangleright$ `[]` $\Rightarrow$ échec
2. `[3;4;5]` $\triangleright$ `(e::reste)` $\Rightarrow$
   `3::[4;5]` $\triangleright$ `(e::reste)` $\Rightarrow$ *OK* `e=3, reste=[4;5]` $\Rightarrow$
   résultat $\Rightarrow$ 3

(2) Appel `(premier [3])` $\Rightarrow$
`3::[]` $\triangleright$ `(e::reste)` $\Rightarrow$ *OK* `e=3`, résultat $\Rightarrow$ 3

Autres appels pour `premier`

```
# let premier l =
match l
with []        -> failwith "premier"
  |  e::reste -> e;;
val premier : 'a list -> 'a = <fun>

# premier [];;
Exception: Failure "premier".
```

Dans la librairie: `List.hd`

Fonctions sur les listes(vide)

La fonction qui teste si une liste est vide:

```
# let vide l =
match l
with []   -> true
   |  _    -> false;;
val vide : 'a list -> bool = <fun>

# vide [1;2];;
- : bool = false

# vide [];;
- : bool = true
```

Fonctions sur les listes(`reste`)

La fonction qui donne le reste d'une liste:

```
# let reste l =
match l
with []    -> failwith "reste"
   |  _::r -> r;;
val reste : 'a list -> 'a list = <fun>

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

# reste [];;
Exception: Failure "reste".
```

Dans la librairie: `List.tl`

Fonctions récursives sur les listes

Correspondent souvent au schéma:

Schéma no. 4: fonction récursive `f(l)` sur une liste *l*

1. Case de base: $l = $ `[]` $\rightarrow$ résultat sans appel récursif;
2. Cas récursif: $l =$ `e::reste` $\rightarrow$ appel récursif `f(reste)` sur une liste plus petite que *l*.

Les appels récursifs se feront sur une liste plus petite que *l* pour garantir la terminaison.

Fonctions récursives sur les listes (exemple)

Compter le nombre d'éléments d'une liste:

```
# let rec longueur l =
   match l
   with [] -> 0
     |  _::reste -> 1 +  longueur reste;;
val longueur : 'a list -> int = <fun>

# longueur [];;
- : int = 0

# longueur ["a"; "salut"];;
- : int = 2
```

Dans la librairie: `List.lenght`

Appel à `longueur`

```
# let rec longueur l =
match l
with [] -> 0
   |  _::reste -> 1 +  longueur reste;;
val longueur : 'a list -> int = <fun>
```

Appel `longueur [1;5;2]`
⇒ le 2ème motif réussit avec `reste=[5;2]`
⇒ `1 + (longueur [5;2])`
⇒ `1 + (longueur 5::[2])`
⇒ `1 + 1 + (longueur [2])`
⇒ `1 + 1 + (longueur 2::[])`
⇒ `1 + 1 + 1 + longueur []` (ici, c'est le 1er motif)
⇒ `1 + 1 + 1 + 0`

Fonction `appartient`

Teste si un élément appartient à une liste:

```
# let rec appartient e l =
match l
with [] -> false
   |  a::reste -> e=a || appartient e reste;;
 val appartient : 'a -> 'a list -> bool = <fun>

# appartient 1 [5;6;3];;
- : bool = false

# appartient 1 [5;3;1;6];;
- : bool = true
```

Dans la librairie: `List.mem`

Appel à `appartient`

```
# let rec appartient e l = match l
   with [] -> false
     |  a::reste -> e=a || appartient e reste;;
```

```
appartient 1 [5;6;3] ⇒
appartient 1 (5::[6;3]) ⇒
1 = 5 || (appartient 1 [6;3]) ⇒
false || (appartient 1 (6::[3])) ⇒
1 = 6 || (appartient 1 [3]) ⇒
false || (appartient 1 (3::[])) ⇒
1 = 3 || (appartient 1 []) ⇒
false || (appartient 1 []) ⇒
false
```

Fonctions sur les listes avec nouvelle liste en résultat

Problème:

1. Parcours d'une liste
2. extraire certains éléments,
3. fabriquer avec $\Rightarrow$ une nouvelle liste (résultat).

Fonction `remplace`

Remplace par *x* la première occurrence de *e* dans une *nouvelle liste* construite lors des appels.

Renvoie la liste inchangé si *e* n'est pas trouvé:

```
# remplace 'a' 'b' ['e'; '2'; 'a'; '?'];;
- : char list = ['e'; '2'; 'b'; '?']

# remplace 1 6 [2;3;4];;
- : int list = [2; 3; 4]
```

Fonction `remplace`

remplace(e,x,l)=

1. si l=[] $\rightarrow$ rien à remplacer, on renvoie [],
2. si l=a::reste $\rightarrow$
   1. si a=e $\rightarrow$ on construit une liste avec x à la place de *e* (en premier élément) et reste en suite de la liste.
      On a terminé: $\Rightarrow$ `a::reste`
   2. si a $\neq$ e $\rightarrow$ pas de remplacement. On «remet» *a* en premier élément et en suite de liste, le résultat de l'appel récursif sur le reste:
      `a::(remplace e x reste)`

En Ocaml?

Fonction `remplace`

Remplace par *x* la première occurrence de *e* dans une *nouvelle liste* construite lors des appels.

Renvoie la liste inchangé si *e* n'est pas trouvé:

```
# let rec remplace e x l = match l
   with [] -> []
    |  a::reste -> if e=a then x::reste
                   else a::(remplace e x reste)
val remplace : 'a -> 'a -> 'a list -> 'a list = <fun>
```

Déroulement d'un appel

On «remet» chaque élément différent de celui cherché dans un nouvelle liste construite progressivement lors des appels

```
remplace 'a' 'b' ['e'; '2'; 'a'; '?';'h'] ⇒
remplace 'a' 'b' ('e'::[ '2'; 'a'; '?';'h']) ⇒
'a' ≠ 'e' ⇒
```

on remet `'e'` dans la liste et on fait un appel sur le reste

```
'e'::(remplace 'a' 'b' ['2'; 'a'; '?';'h']) ⇒
'a' ≠ '2' ⇒
'e'::('2'::(remplace 'a' 'b' ['a'; '?';'h'])) ⇒
'a' = 'a' ⇒
```

FIN: on met dans la liste l'élément nouveau `'b'`

```
'e'::('2'::('b'::['?';'h']))
= ['e'; '2'; 'b'; '?';'h']
```

Les fonctions de la librairie Ocaml

- Nombreuses et organisées en *modules*,
- accessibles avec la *notation pointée*,

Exemple: le module `List` contient des utilitaires sur les listes. Pour utiliser la fonction `length` qui calcule la longueur d'une liste, on écrira:

```
# List.length ["ab"; "bonjour"];;
- : int = 2
```

Quelques modules de la librairie

- Module Arg: parsing of command line arguments
- Module Array: array operations
- Module Buffer: extensible string buffers
- Module Callback: registering Caml values with the C runtime
- Module Char: character operations
- Module Complex: Complex numbers
- Module Format: pretty printing
- Module Gc: memory management control and statistics
- Module Genlex: a generic lexical analyzer
- Module Hashtbl: hash tables and hash functions
- Module Int32: 32-bit integers
- Module Int64: 64-bit integers
- Module List: list operations
- Module Map: association tables over ordered types
- Module Printf: formatting printing functions

Quelques fonctions du module `List`

- `val length : 'a list -> int`
  Return the length (number of elements) of the given list.
- `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.
- `val nth : 'a list -> int -> 'a`
  Return the n-th element of the given list. The first element (head of the list) is at position 0.
- `val rev : 'a list -> 'a list`
  List reversal.
- `val append : 'a list -> 'a list -> 'a list`
  Catenate two lists. Same function as the infix operator @.

Fonctionnelles

Fonction d'ordre supérieur ou fonctionnelle

Fonction qui *prend des fonctions en argument* ou qui les retourne en résultat.

Fonctionnelle `map`: prend en arguments

- une fonction f,
- une liste l ,
- construit une *nouvelle liste*, avec tous les résultats d'application de f aux éléments de l

$$\texttt{map}\ \ f\ \ [a_1; a_2; \ldots a_n] \Rightarrow [f(a_1); f(a_2); \ldots f(a_n)]$$

to map: appliquer partout...

Fonctionnelle List.map

Arguments:

- une fonction *f*, et une liste *l* ,
- construit une *nouvelle liste*, avec tous les résultats d'application de *f* aux éléments de *l*

$$\texttt{map}\ f\ [a_1; a_2; \ldots a_n] \Rightarrow [f(a_1); f(a_2); \ldots f(a_n)]$$

Fait partie du module `List`.

```
# let succ x = x+1;;

# List.map succ [1;2;3];;
- : int list = [2; 3; 4]
```

to map: appliquer partout...

Une liste de vols

```
type vol = {dest:string; num:int; prix:float; disp:int};;

let ccs = {dest="Caracas"; num=221; prix=800.0; disp=10};;
let mxc = {dest="Mexico"; num=121; prix=900.0; disp=6};;
let mx1 = {dest="Mexico"; num=654; prix=750.0; disp=10};;
let mx2 = {dest="Mexico"; num=680; prix=750.0; disp=5};;
let ny = {dest="New York"; num=110; prix=400.0; disp=56};;

let lvols = [ccs; mxc;mx1;mx2; ny];;
val lvols : vol list = ...
```

Extraire la liste de numéros de vols

Problème: à partir d'une liste de vols, utiliser `List.map` pour extraire la liste de numéros de vols.

Solution: appliquer `map` sur la liste avec la fonction:

- `donneNumVol`: prend un vol et renvoie son numéro;

```
# let donneNumVol v = v.num;;

# donneNumVol {dest="New York"; num=110;
               prix=400.0; disp=56};;
-: int = 110
```

Extraire la liste de numéros de vols

`donneNumVol`: appliquée sur un vol extrait son numéro;

```
let donneNumVol v = v.num;;
```

Fonction qui "mappe" (applique partout) `donneNumVol` sur une liste de vols.

```
# let listeNumVols l = List.map donneNumVol l;;
val listeNumVols : vol list -> int list = <fun>

# let lnv = listeNumVols  lvols;;
 val lnv : int list = [221; 121; 654; 680; 110]
```

Fonctionnelle map (code)

Une implantation de map:

```
# let rec map f l =
  match l
  with []      -> []
  |  a::reste -> (f a)::(map f reste);;

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

Expliquez ce type?

Fonctionnelle map (type)

```
# let rec map f l =
  match l
  with []      -> []
  |  a::reste -> (f a)::(map f reste);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `('a -> 'b)`: fonction à appliquer sur chaque élément de la liste, de type `'a`, pour obtenir un résultat de type `'b`
- `'a list`: type de la liste donnée en entrée
- `'b list`: type de la liste renvoyée en résultat

Autre écriture de `listeNumVols`

Avec en paramètre une fonction anonyme (écrite par filtrage):

```
# let listeNumVols l = map (fun {num=n} -> n) l;;
val listeNumVols : vol list -> int list = <fun>

#listeNumVols lvols;;
- : int list = [121; 221; 680; 110; 654]
```

Avec paramètre fonctionnel nommé:

```
# let donneNumVol v = v.num;;

# let listeNumVols l = map donneNumVol l;;
val listeNumVols : vol list -> int list = <fun>
```

Fonctionnelle `filter`

Prend en argument une fonction de test `cond` et une liste, et construit une nouvelle liste composée de tous les éléments de la liste qui satisfont le test *cond*

$$\texttt{filter cond } [a_1; a_2; \ldots a_n] \Rightarrow [a_i \mid \texttt{cond}(a_i) = \texttt{true}]$$

```
# let rec filter cond l =
  match l
  with []      -> []
  |  a::reste -> if (cond a) then a::(filter cond reste)
                 else filter cond reste;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

`filter c l`: filtrer l selon le test c...

Utiliser `filter`: nombres pairs d'une liste

Problème: extraire les nombres pairs d'une liste.

Solution: appliquer `filter` sur la liste avec la fonction de test `est_pair`.

```
# let est_pair x = x mod 2 = 0;;

# let nbPairs l = List.filter est_pair l;;
val nbPairs : int list -> int list = <fun>

# nbPairs [1;2;5;6;80];;
- : int list = [2; 6; 80]
```

- la fonction `est_pair` teste si un entier est pair;
- `filter est_pair` filtre tous les nombres pairs de la liste.

Utiliser `filter`: tous les vols pour Mexico

La fonction `filtreDest` utilise `filter` pour extraire les vols avec une destination donnée.

```
# let filtreDest d lv =
      filter (fun v -> (v.dest = d)) lv;;
val filtreDest : string -> vol list -> vol list = <fun>

# filtreDest "Mexico" lvols;;
- : vol list =
[{dest = "Mexico"; num = 121; prix = 900.; dispo = 6};
{dest = "Mexico"; num = 680; prix = 750.; dispo = 5};
{dest = "Mexico"; num = 654; prix = 750.; dispo = 10}]
```

Autre écriture de filtreDest

La fonction locale `chercheDest` est écrite par filtrage sur son argument enregistrement:

```
# let filtreDest d lv =
    let chercheDest {dest=ville} = (ville = d)
  in List.filter chercheDest lv;;
val filtreDest : string -> vol list -> vol list = <fun>

# filtreDest "Mexico" liste_vols;;
- : vol list =
[{dest = "Mexico"; num = 121; prix = 900.; dispo = 6};
 {dest = "Mexico"; num = 680; prix = 750.; dispo = 5};
 {dest = "Mexico"; num = 654; prix = 750.; dispo = 10}]
```

`filter` est dans le module `List`.

Fonctionnelles du module `List` (extraits du manuel)

- `map: ('a -> 'b) -> 'a list -> 'b list`
- `for_all : ('a -> bool) -> 'a list -> bool` Teste si une condition est vrai pour tous dans la liste.
- `exists: ('a -> bool) -> 'a list -> bool` Teste si une condition est vrai au moins pour un dans la liste.
- `mem: 'a -> 'a list -> bool` fonction `appartient`.
- `filter: ('a -> bool) -> 'a list -> 'a list`
- `find: ('a -> bool) -> 'a list -> 'a`
  `find p l` renvoie le premier élément qui satisfait la condition p.
- `partition:('a->bool)->'a list->'a list* 'a list`
  `partition p l` renvoie une paire de listes (l1, l2), où l1 contient les élément qui satisfont p, et l2 ceux qui ne la satisfont pas.

Les listes d'association

liste d'association: ensemble de couples (cle_i, $valeur_i$) correspondant à la définition "par extension" d'une fonction

- à chaque cle_i correspond $valeur_i$;
- tous les cle_i sont distincts;

```
# let participation = [("Lyon", 51.2); ("Paris", 54.5);
                       ("Neuilly", 43.4)];;
val participation : (string * float) list = ...
```

`participation` associe un nom de ville avec son taux de participation aux éléctions ("fonction" de participation).

Les listes d'association

Exemples:

- la listes des (uid, pwd) associant un mot de passe *pwd* par nom d'utilisateur *uid*;
- la liste des (mot, def) associant une définition par mot du dictionnaire;
- la liste des $(isbn, fiche)$ associant une fiche descriptive par référence de livre *isbn*.

Module List: plusieurs fonctions sur les listes d'associations polymorphes.

Fonction d'association

Permet d'obtenir l'image associée à une clé dans une liste d'association:

```
# let rec assoc x l = match l
  with  []                  -> failwith "assoc"
    |  (cle,info)::reste ->
             if cle=x then info else assoc x reste;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>

# assoc "Paris" participation;;
 - : float = 54.5
```

Utilisable sur des listes d'association de type quelconque.

Dans la bibliothèque: `List.assoc`

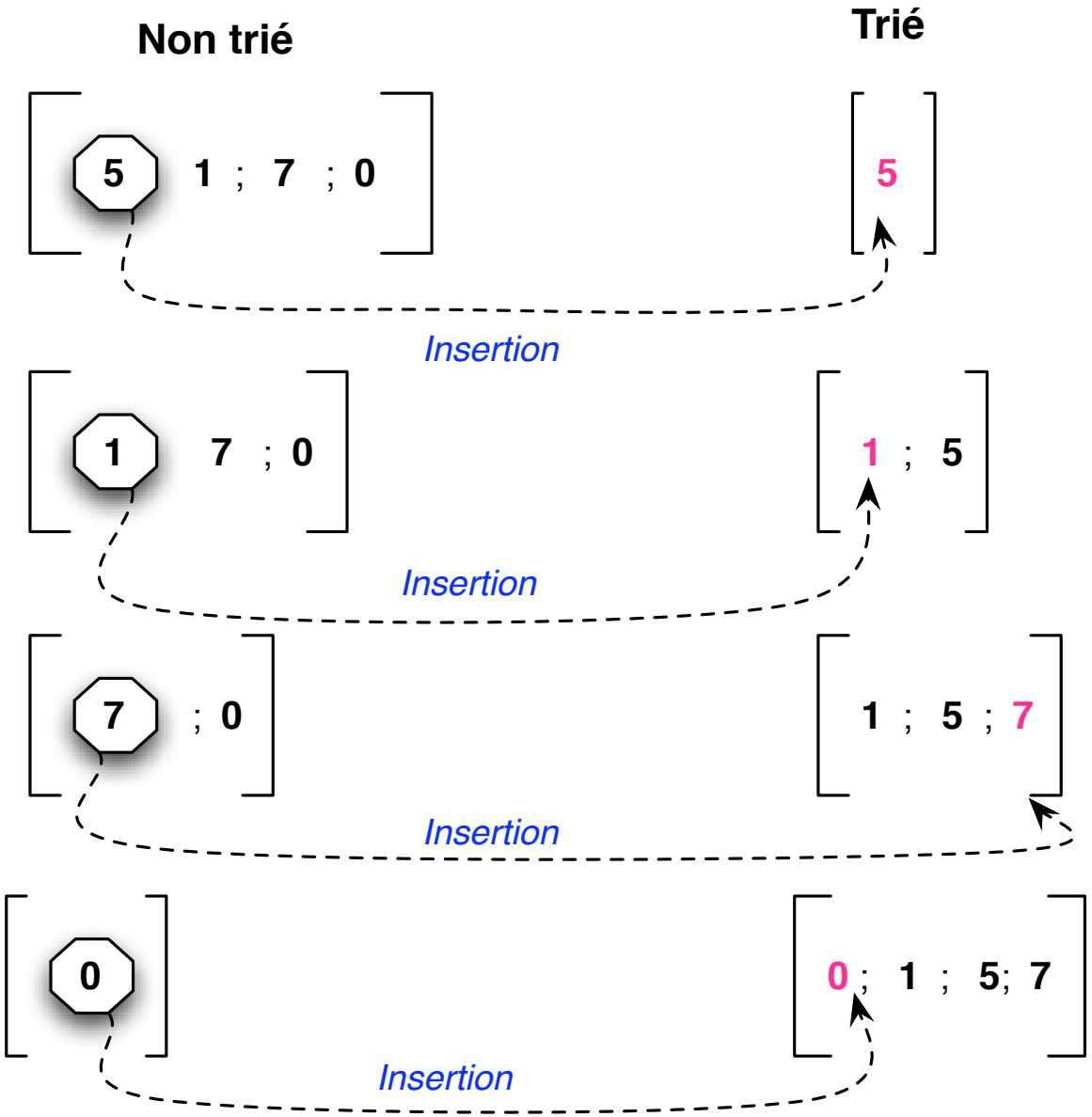
Tri par insertion d'une liste

Partitionner la liste en deux parties: triée et non triée

1. au départ: partie-triée = vide;
2. prendre un élément de partie-non-triée;
   l'insérer à sa place dans partie-triée,

   $\Rightarrow$ partie-triée a un nouvel élément;
3. quand tous dans partie-non triée ont été insérés,
   $\Rightarrow$ partie-triée = toute la liste (Fin)

Tri par insertion d'une liste



La fonction d'insertion

- insère un élément *x* *à la bonne place*
- dans une liste que *l'on suppose triée*.
- Après insertion la liste résultat *est toujours triée*.

La fonction d'insertion

- Si *x est plus petit que a* (le premier la liste), $\Rightarrow$ liste où *x se place juste devant a*.
- Sinon, chercher récursivement où insérer *x* dans le reste,
- $\Rightarrow$ liste qui *commence par a et suit avec le reste de la liste où l'on insère x*.

```
# let rec insere x l = match l
with [] -> [x]
  |  a::reste -> if x<=a then x::a::reste
                 else a::(insere x reste)
val insere : 'a -> 'a list -> 'a list = <fun>

# insere 3 [1;2;7];;
- : int list = [1; 2; 3; 7]
```

La fonction de tri

- si la liste *l* est vide, on la renvoie telle quelle,
- sinon $l = a :: r$. On trie *r*, puis on y insère *a* à la bonne place dans cette partie triée.

```
# let rec tri_insertion l = match l
with [] -> []
  |  a::reste -> insere a (tri_insertion reste);;
val tri_insertion : 'a list -> 'a list = <fun>

# tri_insertion [5;1;7;0];;
- : int list = [0; 1; 5; 7]

#tri_insertion ["salut"; "bonjour"; "adieu"; "arrivedercci"
- : string list=["adieu";"arrivedercci";"bonjour";"salut"]
```

Tri par insertion d'une liste d'entiers(fin)

La fonction `insere` est une fonction auxiliaire de `tri_insertion`.
On les définit ensemble:

```
#let rec insere x l =
     match l
     with [] -> [x]
       |  a::reste -> if x<=a then x::a::reste
                      else a::(insere x reste)
and tri_insertion l =
  match l
  with [] -> []
    |  a::reste -> insere a (tri_insertion reste);;
val insere : 'a -> 'a list -> 'a list = <fun>
val tri_insertion : 'a list -> 'a list = <fun>
```

Déroulement d'un appel

```
tri_insertion [5;1;7;0] ⇒
insere 5 (tri_insertion [1;7;0]) ⇒
insere 5 (insere 1 (tri_insertion [7;0])) ⇒
insere 5 (insere 1 (insere 7 (tri_insertion [0]))) ⇒
insere 5 (insere 1 (insere 7 (insere 0 (tri_insertion
[])))) ⇒
insere 5 (insere 1 (insere 7 (insere 0 ([])))) ⇒
insere 5 (insere 1 (insere 7 ([0]))) ⇒
insere 5 (insere 1 (0::(insere 7 []))) ⇒
insere 5 (insere 1 (0::([7]))) ⇒
insere 5 (insere 1 [0;7]) ⇒
insere 5 (0::(insere 1 [7])) ⇒
insere 5 (0::(1::[7])) ⇒
insere 5 ([0;1;7]) ⇒
0::(insere 5 [1;7]) ⇒
0::(1::(insere 5 [7])) ⇒
0::(1::5::[7]) ⇒ [0; 1; 5; 7]
```

Insertion pour n'importe quel ordre

- La fonction `insere` n'est adéquate que pour les listes de types primitifs;
- autrement dit, ceux que l'on peut comparer par $\leq$;
- Comment faire pour généraliser à tout type de liste?
- Comment faire pour généraliser à tout type d'ordre?

Insertion pour n'importe quel ordre

- passer en argument une fonction `ordre`;
- remplacer le test `(x<=a)` dans `insere`
- par un appel: `(ordre x a)`,
- où `(ordre x a)` $\Rightarrow$ vrai,
  si `x` est "plus petit" que `a` selon cet ordre.

Insertion pour n'importe quel ordre

```
#let rec insere x ordre l =
   match l
   with [] -> [x]
    |  a::reste -> if (ordre x a) then x::a::reste
                   else a::(insere x ordre reste);;
val insere :'a ->('a -> 'a -> bool) -> 'a list -> 'a list =
```

Un appel

Un opérateur infixe *op* mis entre parenthèses (*op*) peut être utilisé comme une “fonction”:

- `(>)` : devient la fonction qui prend a et b et teste si $a > b$;
  $\Rightarrow$ insertion en ordre “plus grand en premier” (décroisant)
- `(<=)` : devient la fonction qui prend a et b et donne vrai si $a \leq b$;
  $\Rightarrow$ insertion en ordre croissant

```
# insere 2 (>) [9;1];;
- : int list = [9; 2; 1]

# insere 2 (<=) [1;9;10];;
- : int list = [1; 2; 9; 10]
```

Déroulement d'un appel

```
#let rec insere x ordre l =
  match l
  with [] -> [x]
    |  a::reste -> if (ordre x a) then x::a::reste
                   else a::(insere x ordre reste);;
val insere : 'a -> ('a -> 'a -> bool) -> 'a list -> 'a list
```

Appel `(insere 2 (>) [9;1])` $\Rightarrow$

`if (>) 2 9 then ... else 9::(insere 2 [1])` $\Rightarrow$

`9::(if (>) 2 1 then 2::(insere 1 []))` $\Rightarrow$

`9::(2::([1]))`

Tri par insertion généralisé

```
#let rec insereGen x ordre l =
  match l
  with [] -> [x]
     |  a::reste ->
           if (ordre x a) then x::a::reste
           else a::(insereGen x ordre reste)
and triInsertionGen ordre l =
  match l
  with [] -> []
     |  a::reste ->
           insereGen a ordre (triInsertionGen ordre reste);;
val insereGen : 'a -> ('a -> 'a -> bool) -> 'a list -> 'a l
val triInsertionGen : ('a -> 'a -> bool) -> 'a list -> 'a l
```

Tri par insertion généralisé (suite)

On peut désormais trier par ordre croissant . . .

```
# triInsertionGen (<=) [3;9;2;5]  ;;
- : int list = [2; 3; 5; 9]
```

. . . ou décroissant:

```
#triInsertionGen (>) [3;9;2;5]  ;;
- : int list = [9; 5; 3; 2]
```

*N*ote: un opérateur infixe (p.e >), devient préfixe (et donc utilisable comme une fonction quelconque) s'il est entouré de parenthèses.

Tri de vols

Nous voudrions effectuer deux sortes de tri:

- par ordre alphabétique des destinations
  $\Rightarrow$ fonction pour tester cet ordre: `ppe_dest`

- par ordre croissant des prix et décroissant dans les disponibilités
  $\Rightarrow$ fonction pour tester cet ordre: `ordre_prix_dispo`

Tri par ordre alphabétique des destinations

La fonction qui teste l'ordre pour deux vols:

```
#let ppe_dest {dest=d1} {dest=d2} = d1 <= d2;;
val ppe_dest : vol -> vol -> bool = <fun>
```

La fonction de tri:

```
#let triDestinations l = triInsertionGen ppe_dest l;;
val triDestinations : vol list -> vol list = <fun>
```

Tri par prix et disponibilité

La fonction qui teste l'ordre pour deux vols:

```
#let ordre_prix_dispo v1 v2 =
    (v1.prix <= v2.prix) && (v1.dispo >= v2.dispo) ||
    (v1.prix < v2.prix);;
val ordre_prix_dispo : vol -> vol -> bool = <fun>
```

La fonction de tri:

```
#let triParPrixEtDispo l =
    triInsertionGen ordre_prix_dispo l;;
val triParPrixEtDispo : vol list -> vol list = <fun>
```

Trier par ordre des destinations

```
#let liste_vols = [mxc; ccs; mxc2; ny; mxc1;
  {dest = "New York"; num = 114; prix = 400.; dispo = 80};
  {dest = "New York"; num = 113; prix = 380.; dispo = 35}];

#let triDestinations l = triInsertionGen ppe_dest l;;

# triDestinations liste_vols;;
- : vol list =
[{dest = "Caracas"; num = 221; prix = 800.; dispo = 10};
 {dest = "Mexico"; num = 121; prix = 900.; dispo = 6};
 {dest = "Mexico"; num = 680; prix = 750.; dispo = 5};
 {dest = "Mexico"; num = 654; prix = 750.; dispo = 10};
 {dest = "New York"; num = 110; prix = 400.; dispo = 56}]
```

Trier par prix et disponibilité

```
#let triParPrixEtDispo l =
   triInsertionGen ordre_prix_dispo l;;

 # triParPrixEtDispo liste_vols;;
- : vol list =
[{dest = "New York"; num = 110; prix = 400.; dispo = 56};
 {dest = "Mexico"; num = 654; prix = 750.; dispo = 10};
 {dest = "Mexico"; num = 680; prix = 750.; dispo = 5};
 {dest = "Caracas"; num = 221; prix = 800.; dispo = 10};
 {dest = "Mexico"; num = 121; prix = 900.; dispo = 6}]
```

Types inductifs (ou types somme)

Type **inductif** ou **somme**

- composé par la *réunion de plusieurs types* $t_1, \ldots t_n$,
- **et/ou** de différents cas syntaxiques des valeurs dans le type;
- chacun des cas est équipé d'un **constructeur** C_i qui permet la distinction syntaxique d'un cas;
- un cas peut contenir des données internes,
- et utiliser récursivement le type définit.

Nommés aussi: *types variants*.

Usage des types somme

Avec eux on peut définir:

1. **types énumeration** de constantes, p.e : les jours de la semaine: `lundi, mardi, ...`
2. **types union**:
   - "unifier" plusieurs types différents (ex: type nombre avec int et float);
   - distinguer plusieurs cas d'un même type (ex: plusieurs unités de mesure toutes données en float)
   - où les valeurs de chaque cas sont distinguées par le constructeur qui l'accompagne.
3. **structures des données récursives**:
   les listes, les arbres, etc.

Types somme: syntaxe

$$\begin{array}{lcl} \texttt{type} < \textit{nom\_type} > & = & \texttt{Constr}_1 \texttt{ of } t_1 \\ & | & \texttt{Constr}_2 \\ & \vdots & \\ & | & \texttt{Constr}_n \texttt{ of } t_n \end{array}$$

- le constructeur $\texttt{Constr}_i$ permet de distinguer syntaxiquement chaque cas des valeurs du type;
- le symbole `|`, prononcé "ou", sépare les cas;
- un constructeur est:
  - un identificateur qui commence par une majuscule;
  - il est constant s'il est tout seul (ex: $\texttt{Constr}_2$)
  - il est paramétré si accompagné par un type t_i (`Constr of` t_i)

(1) Énumérations

Un type qui énumère les couleurs primaires.

```
# type couleur_primaire = Rouge
                        | Vert
                        | Bleu;;
type couleur_primaire = Rouge | Vert | Bleu
```

Une `couleur_primaire` *est soit* `Rouge`*, soit* `Vert`*, soit* `Bleu`.

- Ici, uniquement des constructeurs constants,
- $\Rightarrow$ énumeration des constantes appartenant au type: `Rouge`, `Vert`, et `Bleu`.

Construire une valeur de type somme

Créer une donnée d'un type somme

Utilise nécessairement un des constructeurs C_i déclaré pour le type:

- *seul* s'il est constant;
- suivi d'une *valeur de type* t_i s'il est déclaré avec un paramètre.

Créer une valeur de type `couleur_primaire`:

```
# let r = Rouge;;
val r : couleur_primaire = Rouge
```

Créer une liste de couleurs primaires:

```
# let couleurs = [Rouge;Vert];;
val couleurs : couleur_primaire list = [Rouge; Vert]
```

Fonctions sur un type somme

On les écrit par *filtrage sur tous les cas du type*:

Tester si une couleur est bleu:

```
# let est_bleu c =
match c
with Bleu -> true
 |    _   -> false;;
val est_bleu : couleur_primaire -> bool = <fun>

# let r = Rouge;;
val r : couleur_primaire = Rouge

# est_bleu r;;
- : bool = false
```

(2) Somme de types différents

Le type `num` est la réunion des types `int` et `float`:

- chaque cas est distingué par un constructeur différent,
- qui prend chacun en argument une valeur d'un de ces deux types.

```
type num = Entier of int | Reel of float;;
```

(2) Somme de types différents

```
type num = Entier of int | Reel of float;;
```

Créer une donnée de type `num`:

- utiliser le constructeur `Entier` puis une valeur de type `int`,
- ou le constructeur `Reel` suivi d'une valeur de type `float`.

```
# let trois = Entier 3;;
val trois : num = Entier 3

# let w = Reel 3.4;;
val w : num = Reel 3.4

# let z = Entier 3.4;;
This expression has type float but is here used with type i
```

Additionner deux `num`

`somme_num` permet d'additionner deux valeurs de type `num`:

```
let somme_num (x,y) =
 match (x,y) with
   (Entier m, Entier n) -> Entier (m+n)
| (Entier m, Reel n) -> Reel (n+. (float_of_int m))
| (Reel m, Entier n) -> Reel (m+. (float_of_int n))
| (Reel m, Reel n) -> Reel (m+. n);;

val somme_num : num * num -> num = <fun>


# somme_num ((Entier 2) , (Reel 5.3));;
- : num = Reel 7.3
```

Type carte

Ce type mélange des constructeurs constants et paramétrés:

```
# type carte = As of couleur
             | Roi of couleur
             | Dame of couleur
             | Valet of couleur
             | Simple of couleur * int
             | Joker
and couleur = Pique | Coeur | Carreau | Trefle

# let roi_pique = Roi (Pique);;
roi_pique : carte = Roi Pique

# let sept_coeur = Simple(Coeur,7);;
which sept_coeur : carte = Simple (Coeur, 7)

# Joker;;
- : carte = Joker
```

Valeur à la belote

Une fonction qui calcule la valeur d'une carte à la belote:

```
# let valeur_belote (atout, carte) =
 match carte
 with As _           -> 11
    | Roi _          -> 4
    | Dame _         -> 3
    | Valet c        -> if c = atout then  20 else 2
    | Simple (_,10) -> 10
    | Simple (c,9)   -> if c = atout then 14 else 0
    | _ -> 0;;
val valeur_belote : couleur * carte -> int = <fun>

# valeur_belote (Carreau, (Valet Pique));;
- : int = 2

# valeur_belote (Pique,  (Valet Pique));;
- : int = 20
```

(2) Somme des cas d'un même type

Le type `temps` modélise les unités de mesure du temps, *toutes sous forme d'un entier*.

- les constructeurs servent à *encapsuler* la mesure de temps par une étiquette;
- des mesures de nature différente ne pourront pas être mélangées entre-elles;
- ni mélangées à d'autres entiers;

```
# type temps = Jrn of int
               | Hre of int
               | Min of int
               | Sec of int;;
```

Conversion vers minutes

Une liste de "temps":

```
#  let lt = [Jrn 2; Min 50; Hre 5; Sec 125];;
val lt : temps list = [Jrn 2; Min 50; Hre 5; Sec 125]
```

Fonction qui convertit une quantité de temps en minutes:

```
# let vers_minutes t =
   match t with
     Jrn n -> Min (24*60*n)
   | Hre n -> Min (n*60)
   | Min _ -> t
   | Sec n   -> Min (n/60);;
val vers_minutes : temps -> temps = <fun>
```

Exemple de conversion

```
# vers_minutes (Hre 3);;
- : temps = Min 180

#  let lt = [Jrn 2; Min 50; Hre 5; Sec 125];;
val lt : temps list = [Jrn 2; Min 50; Hre 5; Sec 125]

# List.map vers_minutes lt;;
- : temps list = [Min 2880; Min 50; Min 300; Min 2]
```

Conversion vers un string

Fonction qui converit une quantité de temps en chaîne:

```
#  let lt = [Jrn 2; Min 50; Hre 5; Sec 125];;
val lt : temps list = [Jrn 2; Min 50; Hre 5; Sec 125]

# let to_string t =
   match t with
     Jrn n -> (string_of_int n)^" jours"
   | Hre n -> (string_of_int n)^" heures"
   | Min n -> (string_of_int n)^" minutes"
   | Sec n -> (string_of_int n)^"secondes" ;;
val vers_minutes : temps -> temps = <fun>

# List.map to_string lt;;
- : string list = ["2 jours"; "50 minutes";
                   "5 heures"; "125secondes"]
```

Nombres avec complexes (somme + enregistrements)

Un seul type pour tous les nombres, y compris les complexes.

```
type complexe = {re: float; im: float}
and  num = Entier of int
         | Reel of float
         | Complexe of complexe;;
```

Une liste de complexes:

```
# let ln = [Complexe {re=2.0; im=1.0}; Entier 3];;
```

Notation ..and.. pour la définition simultanée de deux types.

Nombres avec complexes (somme + paire de float)

Ici, les complexes sont modélisées par une paire.

```
# type num = Entier of int
           | Reel of float
           | Complexe of float*float;;
```

Une liste de complexes:

```
# let ln = [Complexe(2.0,1.0); Entier 3];;
```

Objets graphiques

Un objet graphique est soit:

- un point,
- un segment entre 2 points,
- le composite de 2 objets graphiques.

```
# type point = {x: float; y: float};;

# type graphicObj =
            Point of point
          | Segment of point * point
          | Composite of graphicObj * graphicObj;;
```

Ce type est récursif! (inductif)

Objets graphiques

Un segment (entre deux points):

```
# let s1 = Segment ({x=1.; y=0.},{x=1.; y=2.});;
val s1 : graphicObj = ...
```

Un objet composé de deux segments:

```
# let c= Composite(Segment({x=1.; y=0.},{x=1.; y=2.}),
                   Segment({x=1.; y=2.},{x=3.; y=2.}));;
val c : graphicObj = ...
```

(3) Types somme récursifs (inductifs)

Le type `graphicObj` est récursif.

Un objet graphique est:

- soit un point;
- soit un segment entre deux points;
- soit un objet composé de deux objets graphiques (récursivité)

```
# type point = {x: float; y: float};;

# type graphicObj =
            Point of point
          | Segment of point * point
          | Composite of graphicObj * graphicObj;;
```

Un composite de composites

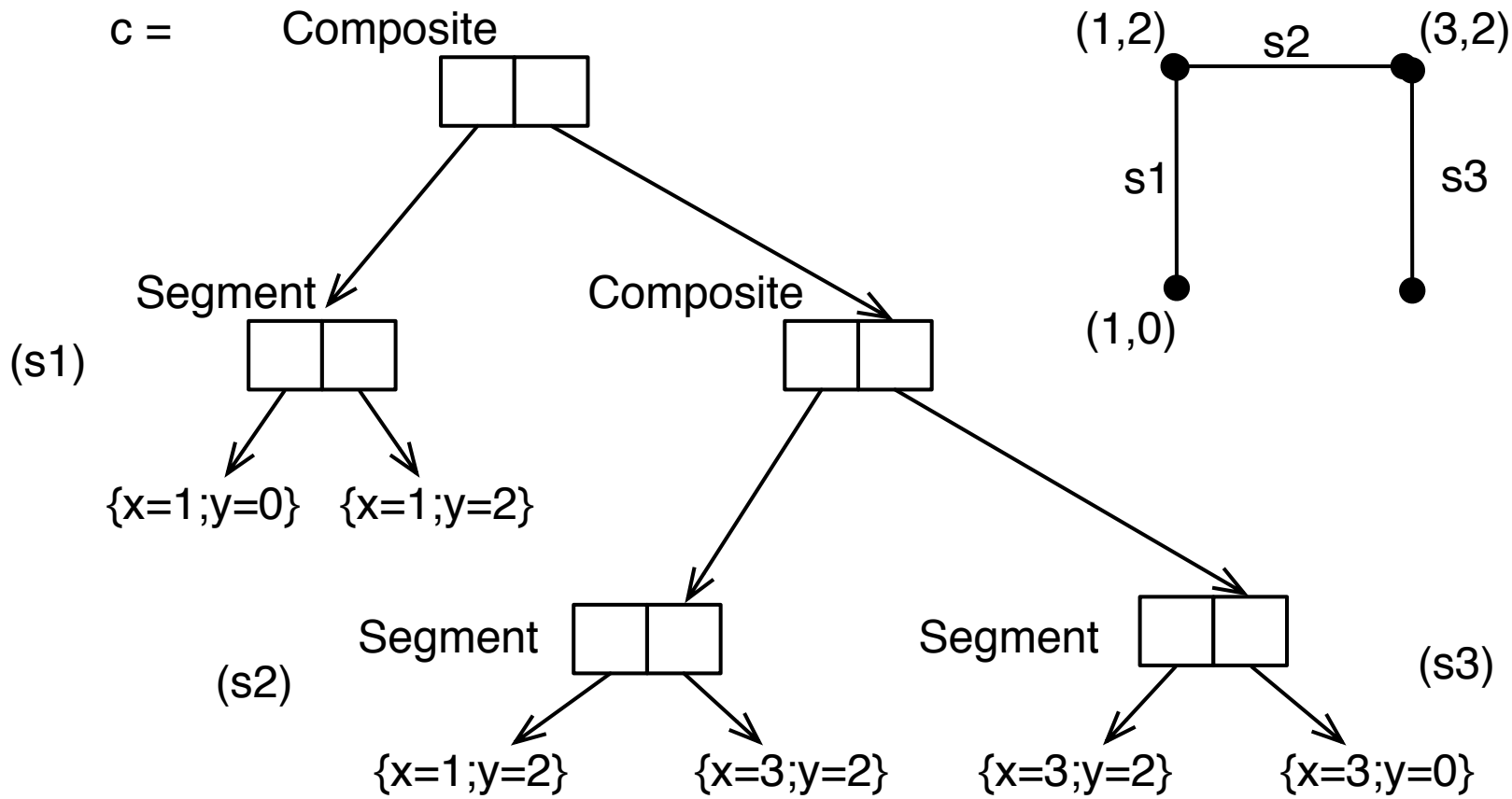
```
 let s1 = Segment ({x=1.; y=0.},{x=1.; y=2.});;
 let s2 = Segment ({x=1.; y=2.},{x=3.; y=2.});;
 let s3 = Segment ({x=3.; y=2.}, {x=3.; y=0.});;

# let c = Composite (s1,
                     Composite (s2, s3));;
```

s1, s2, s3 sont tous de composites.

Un composite de composites

```
let s1 = Segment ({x=1.; y=0.},{x=1.; y=2.});;
let s2 = Segment ({x=1.; y=2.},{x=3.; y=2.});;
let s3 = Segment ({x=3.; y=2.}, {x=3.; y=0.});;
let c1 = Composite (s1,Composite (s2, s3));;
```



Fonctions sur les types récursifs

Les fonctions sur les types inductifs s'écrivent par *filtrage* et *récursivement*.

Compter le nombre de segments d'un objet graphique:

```
# let rec nbSeg o =
 match o
 with  Segment _          -> 1
   | Composite(o1, o2) -> nbSeg o1 + nbSeg o2
   | _                      -> 0;;
val nbSeg : graphicObj -> int = <fun>

# nbSeg c1;;
- : int = 3
```

Les arbres

Arbre

Type de données récursif composé de *noeuds internes*, d'*arêtes* entre deux noeuds, d'une *racine*, et de *feuilles*:

- *racine*: noeud interne d'origine;
- *noeuds internes* : chacun a un ou plusieurs *sous-arbres* ;
- *feuilles*: noeuds terminaux de l'arbre(sans sous-arbres);

Arbres: parents, descendants, etc

- la *racine*: (noeud interne d'origine);
- *noeuds internes*: chacun a un ou plusieurs *sous-arbres* ;
- *feuilles*: noeuds terminaux de l'arbre: ils n'ont pas de sous-arbres;
- les descendants directs d'un noeud sont les noeuds racine de ses sous-arbres;
- p est le parent d'un noeud m si m est un descendant direct de p.
- chaque noeud dans un arbre a au plus un noeud parent;

Utilisation des arbres

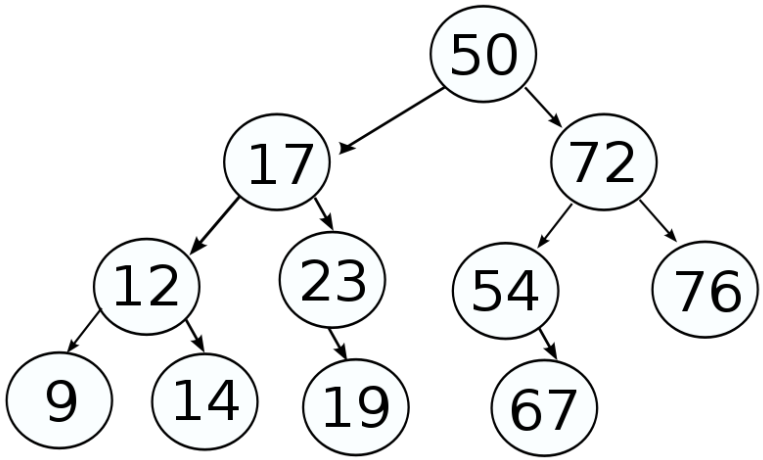
- implantation efficace de parcours de recherche: ordre logarithmique plutôt que linéaire (arbres de recherche);
- représentation de données avec partage (gain de place, dictionnaires) ;
- nombreux algorithmes existants: d'encodage, de compression, décompression, de recherche, etc.
- représentation de données structurées: pages html, xml, ontologies.

Exemple: recherche dans un arbre de recherche

Arbre de recherche tout noeud *n* est tel que:

- tous ses descendants *à gauche sont plus petits* que *n*,
- tous ceux à *droite sont plus grands* que *n*.

Combien de temps pour trouver le noeud 67?
Beaucoup mois que s'ils avaient été rangés linéairement...



Un noeud ici est mal placé, lequel?

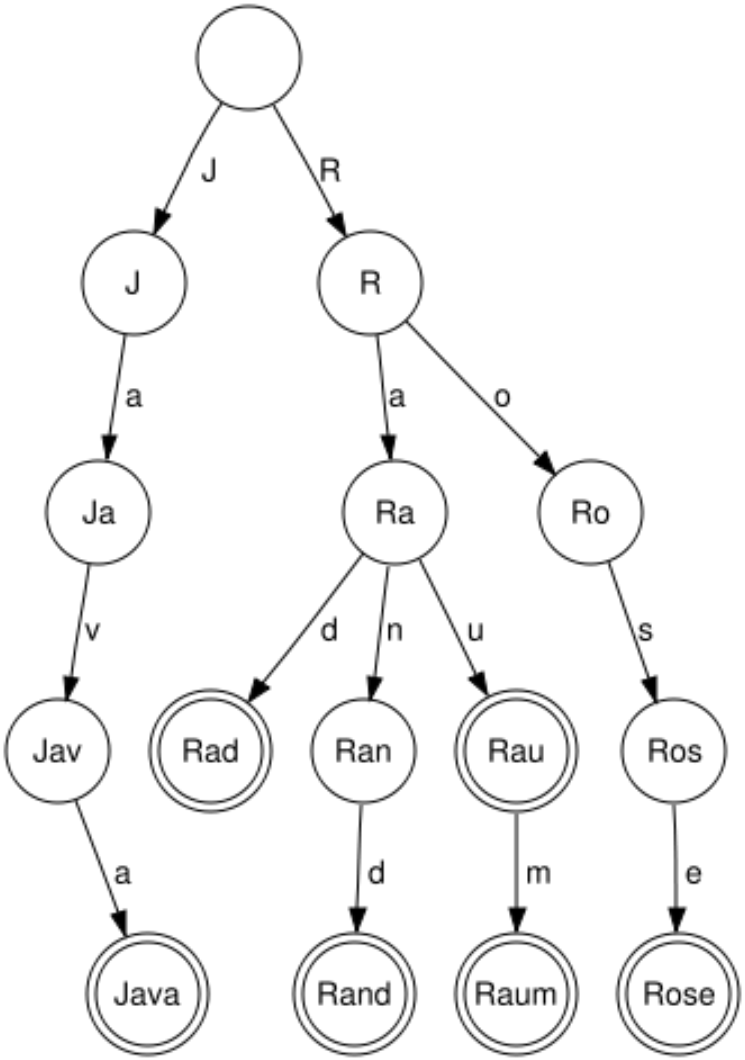
Arbres préfixe

Arbre préfixe: stocker un *dictionnaire de mots* de manière partagée:

- idée: ne représenter qu'une fois les préfixes communs aux mots du dictionnaire;
- le noeud racine est vide;
- un *mot complet est associé à une feuille*;
- il y a un noeud pour chaque *préfixe commun* mais seules les feuilles stockent des valeurs (mots complets);
- la *position* d'un noeud détermine le *préfixe* qui lui correspond: il est formé de tous les caractères "traversés" depuis la racine;
- tous les descendants d'un noeud ont un même préfixe commun;

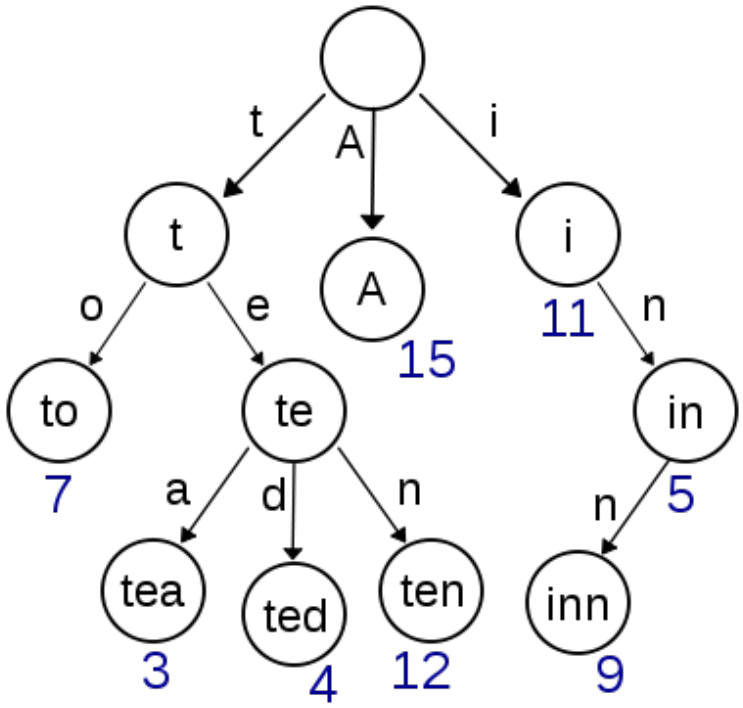
Arbres préfixe

Dictionnaire représenté: *{Java, Rad, Rand, Rau, Raum, Rose}*



Un autre arbre préfixe

Dictionnaire représenté: *{to, tea, ten, inn}*



Les arbres binaires

Arbres binaires pouvant contenir dans chaque noeud une information de type quelconque:

```
# type 'a arbreBin =
        VideBin
      | NoeudBin of 'a * 'a arbreBin * 'a arbreBin;;
```

Un *arbre binaire* est:

- soit *vide*;
- soit un *noeud* composé d'une *donnée* et *deux sous-arbres binaires*.

Instance d'arbres

Un arbre d'entiers:

```
# let a = NoeudBin (1,VideBin, VideBin);;
val a1 : int arbreBin = ...
```

Un arbre de strings:

```
# let b = NoeudBin ("ab",VideBin, VideBin);;
val b : string arbreBin = ...
```

Un arbre de points:

```
# let c = NoeudBin ({x=1.0, y=2.0}, VideBin, VideBin);;
val a : point arbreBin = ...
```

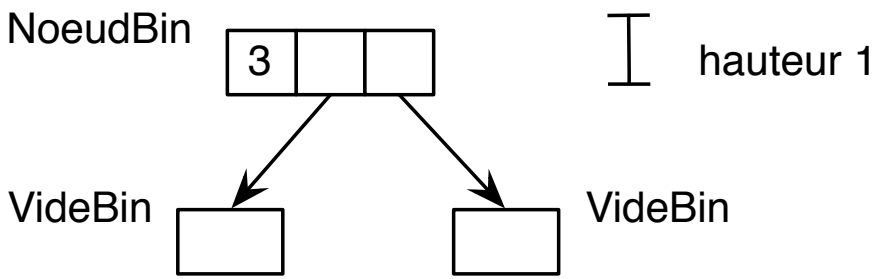
- structure des données arborescente et récursive;
- générique car pouvant contenir des informations de type quelconque.

Hauteur d'un arbre

L'*hauteur* d'un arbre est:

- 0 si l'arbre est vide
- le nombre de noeuds (non vides) de la plus grande branche qui va de la racine jusqu'à un noeud vide.

```
# let a2 = NoeudBin (3,VideBin, VideBin);;
  val a2 : int arbreBin = ...
```



Arbre binaire d'hauteur 2

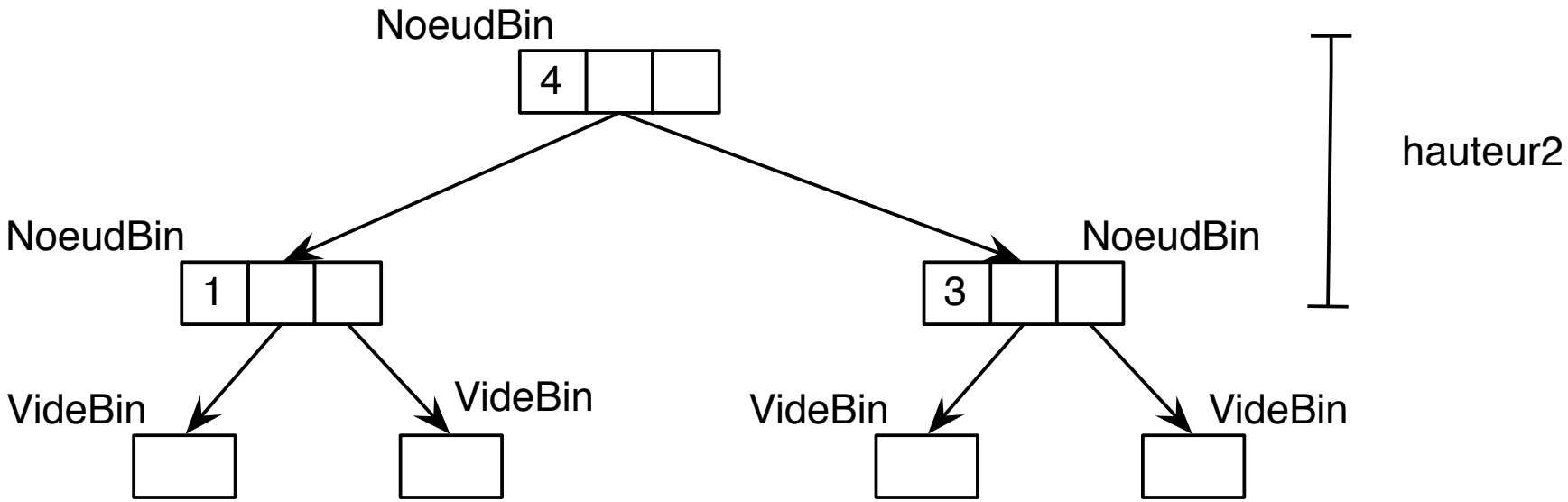
```
# let a1 = NoeudBin (1,VideBin, VideBin);;
# let a2 = NoeudBin (3,VideBin, VideBin);;

# let a = NoeudBin (4, a1, a2);;
val a : int arbreBin = ...
  NoeudBin (4, NoeudBin (1, VideBin, VideBin),
               NoeudBin (3, VideBin, VideBin))
```

Arbre binaire d'hauteur 2

```
# let a1 = NoeudBin (1,VideBin, VideBin);;
# let a2 = NoeudBin (3,VideBin, VideBin);;

# let a = NoeudBin (4, a1, a2);;
val a : int arbreBin = ...
```



Tester l'appartenance à un arbre

```
#let rec btMem e btree =
   match btree with
      VideBin -> false
    | NoeudBin (y, left, right) ->
         e = y || btMem e left || btMem e right;;
val btMem : 'a -> 'a arbreBin -> bool = <fun>
```

- si l'arbre est vide $\Rightarrow$ false
- si le noeud racine contient y et $y = e \Rightarrow$ true
- sinon, on cherche e dans le sous-arbre *gauche*;
- ou on cherche e dans le sous-arbre *droit*;

$\Rightarrow$ Double appel récursif

Appartenance à un arbre

```
#let rec btMem x btree =
   match btree with
      VideBin -> false
    | NoeudBin (y, left, right) ->
         x = y || btMem x left || btMem x right;;
val btMem : 'a -> 'a arbreBin -> bool = <fun>

# btMem 1 a;;
- : bool = true

#  btMem "ab" (NoeudBin ("bc", VideBin, VideBin));;
- : bool = false
```

Cette fonction est polymorphe: elle peut être employé sur tout type d'arbre.

Somme d'éléments d'un arbre

La fonction qui additionne les éléments d'un arbre. Quel est son type?

```
# let rec btAdd bt =
    match bt
    with VideBin              -> 0
     |   NoeudBin (n, a,b) ->
              n + btAdd a  + btAdd b;;
val btAdd : int arbreBin -> int = < fun >

# btAdd a;;
- : int = 8
```

Cette fonction est *monomorphe*: elle ne peut être employé que sur des arbres d'entiers.

Déroulement d'un appel

Cette fonction fait un double appel récursif:

```
# let rec btAdd bt =  match bt
   with VideBin                -> 0
    |   NoeudBin (n, a,b) -> n + btAdd a + btAdd b;;
```

```
btAdd a ⇒
```

```
btAdd(NoeudBin(4,(NoeudBin(1,VideBin,VideBin),
                  (NoeudBin(3,VideBin,VideBin))))) ⇒
  4 + (btAdd (NoeudBin (1, VideBin,VideBin)))
    + (btAdd (NoeudBin (3, VideBin,VideBin))  ⇒
  4 + ( 1 + (btAdd VideBin) + (btAdd VideBin))
      ( 1 + 0 + 0 )
    + ( 3 + (btAdd VideBin) + (btAdd VideBin))⇒
  4 +  (1 + 0 + 0) + (3 + 0 + 0)}  ⇒ 8
```

Arbres, grammaires, langages

Beaucoup d'applications informatiques traitent les entrées fournies par l'utilisateur ou par d'autres programmes.

Parfois ces entrées sont données dans un langage précis:

- petit langage de commandes;
- document structurés, ex: en HTML
- petits langages dédiés: graphique, expressions rationnelles;
- langages de requêtes d'une basse de données;

Les arbres sont parfaits pour représenter de manière interne les "phrases" dans ces langages.

Les expressions arithmétiques

Un exemple classique de "petit langage": les expressions arithmétiques.

```
type aExpr = Cnt of int
           | Var of string
           | Add of aExpr * aExpr
           | Sub of aExpr * aExpr
           | Mul of aExpr * aExpr
           | Div of aExpr * aExpr;;
```

Les expressions arithmétiques sont formés de:

- constantes entières,
- noms de variables (chaînes de caractères);
- opération arithmétiques (+,-,*,/) sur deux sous-expressions.

Expressions constantes

Expression: 2

```
# let deux = Cnt 2;;
val deux : aExpr = Cnt 2
```

Expression: $1 + 2$

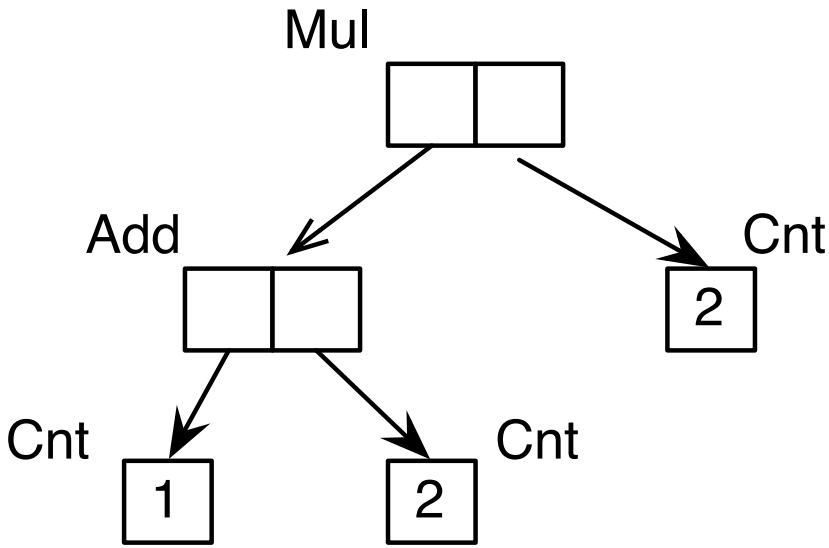
```
# let a = Add((Cnt 1), (Cnt 2));;
val a : aExpr = Add (Cnt 1, Cnt 2)
```

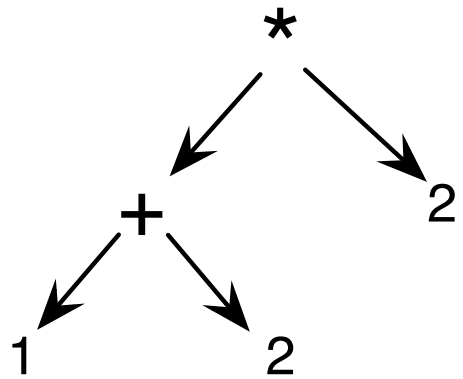
Expression: $(1 + 2) * 2$

```
# let b = Mul(a, deux);;
val b : aExpr = Mul (Add (Cnt 1, Cnt 2), Cnt 2)
```

Expressions constantes

Expression:$(1 + 2) * 2$



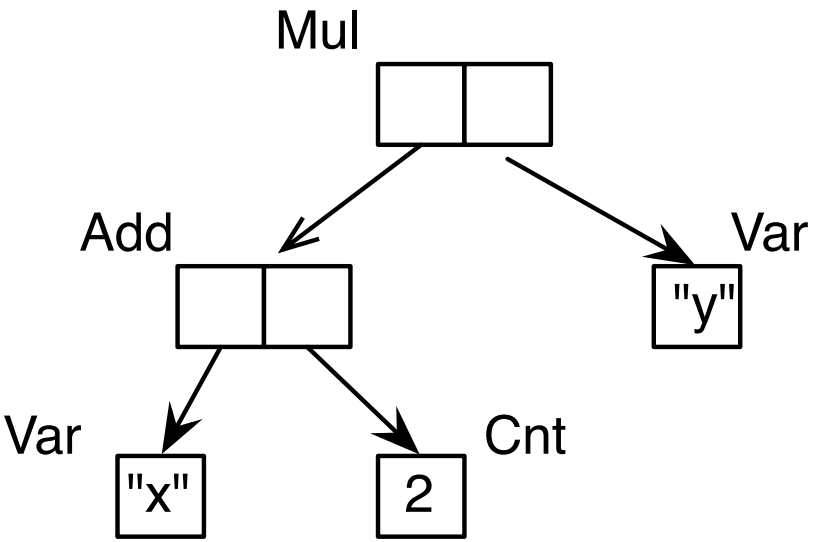


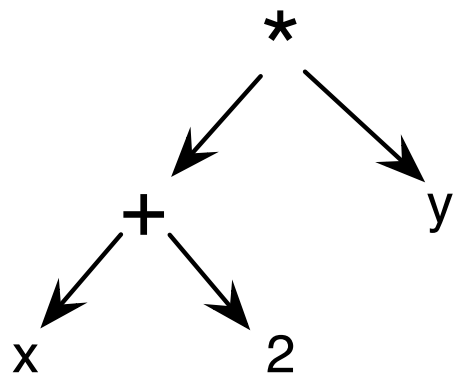
Expressions avec variables

Expression b = $(x + 2) * y$

```
# let a = Add((Var "x"), (Cnt 2));;
val a : aExpr = Add (Var "x", Cnt 2)

# let b = Mul( a, (Var "y"));;
```





Transformer une expression en string

```
# let rec to_string a = match a
  with Cnt n -> string_of_int n
    |   Var x -> x
    |   Add (e1,e2)->(to_string e1)^" + "^(to_string e2)
    |   Mul (e1,e2)->(to_string e1)^" * "^(to_string e2)
    |   Sub (e1,e2)->(to_string e1)^" - "^(to_string e2)
    |   Div (e1,e2)->(to_string e1)^" / "^(to_string e2)
;;
val to_string : aExpr -> string = <fun>

# to_string (Add ((Mul (Var "x", Cnt 3)), Var "y"));;
- : string = "x * 3 + y"
```

Evaluer une expression avec variables

- Pour évaluer une *expression avec variables* $(x + 2) * y$ on doit connaître leurs valeurs.
- La fonction d'évaluation doit prendre en argument un *contexte d'évaluation*: une liste de paires *(nomVar, valeur)*.

Exemple: c = `[("x",5); ("y",2); ("m",6)]` fixe les valeurs des variables x,y,m

Evaluer $(x + 2) * y$ dans le contexte c $\Rightarrow$ 14

Évaluer une expression avec variables

- `env`: contexte d'évaluation (liste d'association);
- `List.assoc`: pour extraire les valeurs des variables.

```
# let rec eval env expr = match expr
  with Cnt n -> n
   |   Var x -> List.assoc x env
   |   Add (e1,e2) -> (eval env e1) + (eval env e2)
   |   Mul (e1,e2) -> (eval env e1) * (eval env e2)
   |   Sub (e1,e2) -> (eval env e1) - (eval env e2)
   |   Div (e1,e2) -> (eval env e1) / (eval env e2)
val eval : string*int list -> aExpr -> int= <fun>

# eval [("x",5); ("y",2); ("m",6)]
       (Add ((Mul (Var "x", Cnt 3)), Var "y"));;
- : int = 17
```

Les arbres de dégré variable

Dégré d'un arbre: nombre de sous-arbres de chaque noeud qui n'est pas terminal.

Les arbres vus en exemple ont un *dégré* constant (2):

- les arbres binaires,
- les expressions arithmétiques,
- les objets graphiques.

Il est intéressant de représenter des arbres dont le nombre de sous-arbres n'est pas prédétermine par son type mais dépend des donnés qui vont être insérés.

Arbres de degré variable: exemple

Exemple: représenter un document structuré en sections, sous-sections, sous-sous-sections:

- chaque section est un noeud de l'arbre;
- les sous-sections de la section sont ses sous-arbres;
- une section peut avoir un nombre arbitraire de sous-sections.

Comment faire?

Type arbre de degré variable

- `TVide`: arbre vide
- `TNoeud`: noeud interne contenant une donnée et une liste de sous-arbres;

```
# type 'a arbre =
    TVide
  | TNoeud of 'a * 'a arbre list;;

let cours =
  TNoeud("Cours Ocaml",
      [TNoeud("Introduction", [TNoeud("Principes", [])]);
       TNoeud("Les types", []);
       TNoeud("Les Fonctions",
           [TNoeud("Déclaration des fonctions", []);
            TNoeud("Typage des fonctions", [])])]);;
val cours : string arbre = ...
```

Compter les noeuds d'un arbre

```
# let rec tree_size t =
    let rec add l = match l
    with  []    -> 0
     |     x::r -> x + (add r)
    in match t
    with TVide  -> 0
     |   TNoeud(_, tlist) ->
              1 + (add (List.map tree_size tlist));;
val tree_size : 'a arbre -> int = <fun>

# tree_size cours;;
- : int = 7
```

Exercices:

- calculer la hauteur d'un arbre.
- Ecrire un map sur les arbres binaires et de degré variable.

Quelques traits impératifs

Ocaml possède plusieurs constructions de programmation impérative:

- variables et structures des données modifiables (tableaux, enregistrements, références, flux),
- affectation,
- instructions d'entrée/sortie,
- boucles itératives et
- exceptions.

Calculs avec expressions

La manière de calculer en programmation fonctionnelle est basée sur l'*utilisation d'expressions*, qui *calculent des valeurs* résultats:

```
# let x = 3 in x + 4;;
```

```
# let f (x) = x + 1 in f(2);;
```

Cette formulation est très proche des mathématiques.

L' exécution aboutit à une valeur *rendue en résultat*.

Calculs avec effets

Une autre manière de calculer, dite *impérative* consiste à:

exécuter des instructions d'entrées/sorties, d'affectation ou de modification de la mémoire. Ces instructions ne rendent pas de résultat, mais changent l'état de la machine: à savoir, de la mémoire ou des entrées/sorties.

Ces actions sont appelés *effets*.

Effets: actions sans résultat

En style impératif, on calcule avec des suites d'instructions, qui *ne rendent pas de résultat*:

```
int x = System.in.read();
int valAbs;
if (x<=0) {valAbs = -x
} else    {valAbs = x
} System.out.write(valAbs);
```

Ce programme *change l'état de la mémoire* (par affectation des variables `x` et `valAbs`) et des entrées/sorties.

En Ocaml on pourrait écrire:

```
let valAbs =
  let x = read_int() in
   if x<=0 then -x  else  x
in print_int(valAbs);;
```

Effets en Ocaml

- Les instructions d'entrée/sorties d'Ocaml, sont des *effets*,
- il n'y a pas de valeur calculée mais un changement de l'état des entrées/sorties.

```
# print_string"Bonjour";;
```

Effets en Ocaml

En Ocaml, même un effet doit rendre un résultat, et donc avoir un type.

```
# print_string"Bonjour";;
Bonjour- : unit = ()
```

Examinons cette réponse:

- `Bonjour` est *l'effet produit* par l'exécution.
- `- : unit = ()` est la réponse d'Ocaml: avec la valeur calculée et son type.

Le type des effets: le type *vide*

En Ocaml:

- Le *résultat d'un effet* est la "valeur vide" noté `()`,
- Le *type d'un effet* est "type vide" noté `unit`.

```
# print_string"Bonjour";;
Bonjour- : unit = ()
```

$\Rightarrow$ Le type *vide* est l'équivalent du type *void* de Java.

Séquences d'instructions

Suite d'instructions séparées par des point-virgules.

- son *résultat* et son *type* sont la *la valeur et le type de la dernière instruction de la séquence*,
- une instruction intermédiaire qui renvoie un résultat autre que `()` est signalée par un avertissement du compilateur.

```
# print_string"Bonjour"; print_string" Adieu";;
Bonjour Adieu- : unit = ()

# 3+4; print_string" Adieu";;
Warning S: this expression should have type unit.
 Adieu- : unit = ()
```

Entrées/sorties en Ocaml

Fonctions d'entrées/sorties standard: dédiées aux canaux standard (clavier et écran)

```
read_line:      unit → string
read_int:       unit → int
print_string:   string → unit
print_int:      int → unit
print_newline:  unit → unit
```

Fonctions d'entrées/sorties

Les canaux d'entrée/sortie sont crées par ouverture de fichiers en utilisant les fonctions:

```
open_in:   string → in_channel
open_out:  string → out_channel
```

Ecriture de la chaîne "hello" sur le fichier `temp`:

```
# let ch = open_out "temp" in
  output_string ch "hello"; close_out ch;;
- : unit = ()

> more temp
hello
```

Fonctions d'entrées/sorties

```
output_char:    out_channel → char → unit
output_string:  out_channel → string → unit
input_char:     in_channel → char
input_line:     in_channel → string
```

Exemples

Une fonction pour afficher un arbre d'entiers:

```
#let rec affiche_int_arbre= function
      Vide -> print_string  "()"
    | Noeud (Vide, y,  Vide) -> print_int y
    | Noeud (g, y,  d) ->
        print_string "("; (affiche_int_arbre g);
        print_string ","; print_int y; print_string ",";
       (affiche_int_arbre d);  print_string ")";;
val affiche_int_arbre : int arbre -> unit = <fun>

# affiche_int_arbre
    Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 5,
    Noeud (Noeud (Vide, 1, Vide), 3, Vide))

((1,3,5),7,(1,3,()))- : unit = ()
```

Un programme intéractif

Calcul de note d'un examen: demande une note d'écrit et d'oral et calcule le résultat final en leur appliquant la fonction `f`:

```
#let calcule_note f =
  print_string"Note d'ecrit: ";
  let n = float_of_string(read_line()) in
  print_string"Note d'oral: ";
  let m = float_of_string(read_line()) in
  f(n,m);;
val calcule_note : (float * float -> 'a) -> 'a = <fun>
```

Un appel avec calcul de moyenne

Calcul sur la base d'une moyenne:

```
#let moyenne () =
  let calc (x,y) =
     let m = (x+.y)/.2.0 in
     print_string
       (if m> 10.0 then  "RECU(E)" else  "AJOURNE(E)");
     print_newline()
  in calcule_note calc;;
 val moyenne : unit -> unit = <fun>

# moyenne();;
Note d'ecrit: 14.3
Note d'oral: 11.5
RECU(E)
- : unit = ()
```

Un autre mode de calcul

Calcul sur la base d'une moyenne pondérée:

```
# let autre_note () =
   calcule_note (fun (x,y) -> (0.6*. x) +. (0.4 *. y));;
val autre_note : unit -> float = <fun>
```

Enregistrements modifiables

Déclarés de manière à rendre (explicitement) des champs modifiables, avec le mot-clé `mutable`.

```
# type point_mutable =
        {mutable x: float;mutable y: float};;

# let translate p dx dy= p.x<-p.x +.dx; p.y<- p.y+.dy;;
val translate : point_mutable -> float -> float -> unit = <

#let monpoint = {x=0.0;y=0.0};;
val monpoint : point_mutable = {x=0; y=0}

#translate monpoint 1.0 2.0;;
 - : unit = ()

# monpoint;;
- : point_mutable = {x=1; y=2}
```

Les tableaux

Les tableaux sont des structures des données *mutables avec type polymorphe*.

- un tableau avec éléments de type `t` est de type `t array`.
- un exemple de tableau `[| 0; 1; 3|]`
- L'affectation d'une cellule se fait via la flèche inversée `<-`.
- Les indices sont des entiers, dont le premier est zéro.

Les tableaux

```
# let a =  [| 0; 1; 3|]  ;;
val a : int array = [|0; 1; 3|]

# a.(0)<-7;;
- : unit = ()

# a;;
- : int array = [|7; 1; 3|]
```

Opérations dans le module `Array`:

- `Array.lenght`,
- `Array.create long init`, crée un tableau de taille `long` initialisé à `init` partout.

Boucle For

```
# let digits = Array.create 10 0;;
val digits : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]

#  for i = 0 to 9 do
       digits.(i) <- digits.(i) + i done;;
- : unit = ()

# for i = 10 downto 0 do print_int i done;;
109876543210- : unit = ()

# let fact (n) =
  let accu = ref 1 in
  for i = 1 to n do accu := i * !accu done;
  !accu;;
val fact : int -> int = <fun>

# fact(2);;
- : int = 2
```

Boucle `While`:

```
#let  j =ref 10 in while (!j)> 0
    do begin print_int !j; j := !j-1; end done;;
10987654321- : unit = ()

# j;;
Characters 0-1:
Unbound value j
```

Notez que j est une variable locale à l'expresion suivant le `in`, et de ce fait, elle est n'est plus definie en dehors de celle-ci.

Les exceptions

Les exceptions permettent de traiter les situations qui empêchent l'accomplissement normal d'un programme.

```
# 2 + (1/0) -3;;
Exception: Division_by_zero.

# List.hd [];;
Exception: Failure "hd".
```

Les exceptions

- Lever une exception, c'est signaler qu'une situation anormale est survenue.
- Cela a pour effet *d'interrompre* l'évaluation normale du programme.
- *Traiter* une exception, c'est répondre à cette situation en exécutant les actions appropriées.
- Idée: permettre *la poursuite du programme de manière normale*.

Si une exception est lévée et non traitée, elle provoque l'arrêt du programme.

Déclaration d'exceptions

L'ensemble des exceptions définies en Ocaml sont regroupés dans un *type somme* spécial appelé `exn`.

Chaque constructeur du type `exn` correspond à une exception connue par Ocaml. Il en existe des pre-définis:

```
type exn =
    ...
    | Division_by_zero
    | Failure of string
    ...
```

Déclaration d'exceptions

Ce type a la particularité d'être *extensible*: l'utilisateur à la possibilité de lui ajouter des constructeurs. Lorsque l'on écrit:

```
exception Erreur_Entier of int;;
```

nous ajoutons un cas à la définition de `exn`:

```
type exn =
    ...
    | Division_by_zero
    | Failure of string
    | Erreur_Entier of int
    ...
```

Création de valeurs exceptionnelles

- Les valeurs de type `exn` ont un comportement normal,
- sauf si elles sont employées avec la fonction `raise` de déclenchement d'exceptions.

```
# let toto = Failure "toto";;
val toto : exn = Failure "toto"

# raise (toto);;
Exception: Failure "toto".
```

Construction `raise`

Exemple: la fonction `premier` sur les listes:

```
# let premier l =
match l
with []       -> raise(Failure "premier")
   |  e::reste -> e;;

# premier [];;
Exception: Failure "premier".
```

Traitement d'exceptions

Le traitement des exceptions en Ocaml se fait par filtrage:

- on tente d'exécuter normalement un programme,
- et dans le cas où des exceptions se produisent, on les "attrape" en les filtrant,
- pour déterminer les actions à effectuer dans chaque cas.

On utilise la construction `try-with`:

```
try e
with p1 -> e1
   |  p2 -> e2
   ...
   |  pn -> en
```

où `p1, ...pn` sont des valeurs de type `exn`.

Traitement d'exceptions

```
try e
with p1 -> e1
   |  p2 -> e2
   ...
   |  pn -> en
```

On tente d'évaluer `e`:

- si l'évaluation se fait normalement en une valeur *v*, alors toute l'expression `try-with` donne *v* en résultat;
- si l'évaluation de `e` lève une exception, on utilise les motifs `p1, .. pn` pour filter cette exception, à la recherche d'un traitement `ei` à exécuter;
- si aucun motif ne filtre la valeur levée, elle est propagée au délà du `try-with`.

Traitement d'exceptions: un exemple

Un cas de traitement:

```
# try 1/0 with Division_by_zero -> 0;;
- : int = 0
```

Un cas sans traitement:

```
# try 1/0 with (Failure _) -> 0;;
Exception: Division_by_zero.
```

Autres exemples

```
#exception ListeVide;;
exception ListeVide

#let tete l =
  match l with
  []    ->raise ListeVide
| hd::tl->hd;;
val tete : 'a list -> 'a = <fun>

#tete [1;2];;
- : int = 1

#tete [];;
Uncaught exception: ListeVide
```

Autres exemples

```
#exception Negatif of int;;
exception Negatif of int

#let rec fact n  if n <0 then raise (Negatif n) else
        if n=0 then 1 else n*fact(n-1);;
 val fact : int -> int = <fun>

#let afficheFactPremier l  =
  try  print_int (fact (tete l))
  with  ListeVide -> print_string "la liste est vide"
    |   Negatif x -> print_int x;print_string " est negatif
    |   _         -> print_string " Autre exception";;
val affichePremier : int list -> unit = <fun>
```

Suite de l'exemple

```
# afficheFactPremier [5;-3;6];;
120- : unit = ()

# afficheFactPremier [-5;-3;6];;
-5 est negatif- : unit = ()

# afficheFactPremier [];;
la liste est vide- : unit = ()
```

Compléments sur le filtrage: les motifs

Les motifs peuvent:

- contenir des variables, qui seront liés à une sous-partie de la valeur filtrée,
- être emboîtés, contenir des produits, des sommes, des enregistrements, listes, etc.,
- ne pas énumérer tous les champs des enregistrements,
- lier une valeur intermédiaire avec mot clé `as`,
- n'imposer aucune contrainte (caractère _)

Motifs emboîtés

```
# type cercle = {origine : int*int; rayon: int}

# let deplace_et_agrandit c a d =
match c
with {origine = (0,0) as o; rayon = r} ->
                {origine = o; rayon = r+d}
  |    {origine = (x,y); rayon = r} ->
                {origine = (x+a, y+a) ; rayon = r+d}
val deplace_et_agrandit : cercle -> int -> int -> cercle =
```

Notez qu'il n'est pas nécessaire d'énumérer tous les champs d'enregistrement et l'utilisation de `as`.

Copie avec `with`

```
# let deplace_et_agrandit c a d =
  let y = {c with rayon = c.rayon + d} in
  match c
  with {origine = (0,0)} -> y
 |     {origine = (x,y); rayon = r} ->
              {origine = (x+a, y+a) ; rayon = r+d}
val deplace_et_agrandit : cercle -> int -> int -> cercle =
```

La notation `let y = {c with rayon = c.rayon + d}` signifie:
soit `y` un enregistrement identique à `c` sauf pour le champ `rayon` qui vaut `c.rayon + d`

Autres exemples

Implication logique:

```
# let et_logique v = match v with
          (true,true) -> true
        |   _         -> false;;
val et_logique : bool * bool -> bool = <fun>
```

Combinaison de motifs:

```
# let est_une_voyelle c = match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false ;;
val est_une_voyelle : char -> bool = <fun>

# est_une_voyelle 'i' ;;
- : bool = true
# est_une_voyelle 'j' ;;
- : bool = false
```

Motifs avec variables

Une variable ne peut pas être liée deux fois dans le même motif.

```
# let deplace_et_agrandit c a d =
  let y = {c with rayon = c.rayon + d} in
  match c
  with {origine = (0,0)} -> y
 |     {origine = (x,x); rayon = r} ->
                {origine = (x+a, x+a) ; rayon = r+d}

 The x variable is bound several times in this matching
```