#Version : bpf.tex 3 sep 24 16:55:07 compiled 3 septembre 2024, 16 h 55 CEST

Bases de la programmation fonctionnelle

Nicolas Bedon

Université de Rouen

Références I

E. Chailloux, P. Manoury, B. Pagano

Developing applications with Objective Caml, O'Reilly, 2000

O. Mallet

Informatique MP2I – Cours, programmes en C et OCaml et exercices corrigés, Ellipses, 2023 ; ISBN : 9782340083400

S. Veigneau

Approche impérative et fonctionnelle de l'algorithmique, Springer, 1999

`http://ocaml.org`

Site de OCaml : téléchargement, documentation, forums, ...

E. Laugerotte

Cours de programmation fonctionnelle

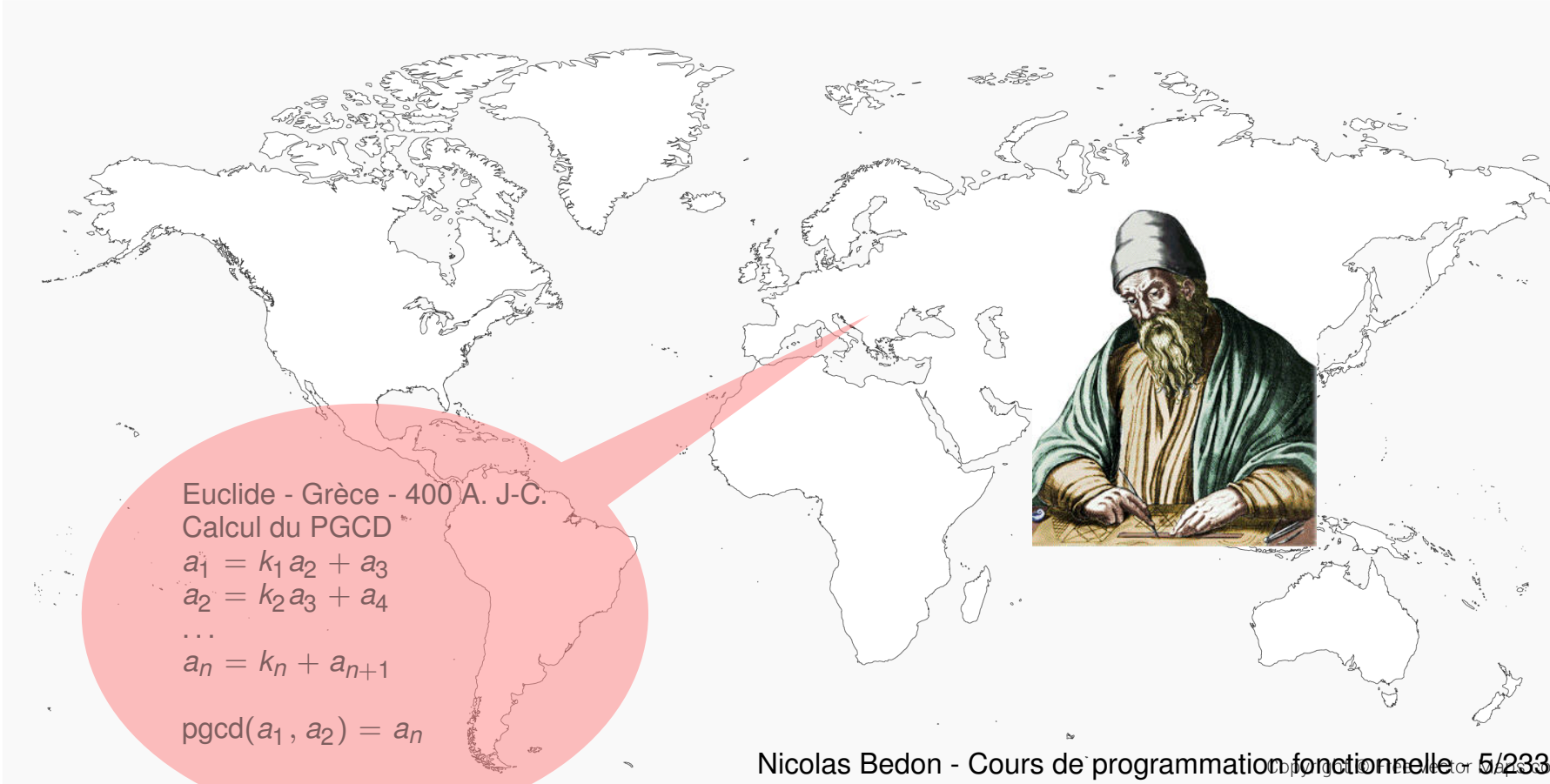
Dépendances

Très peu de pré-requis.

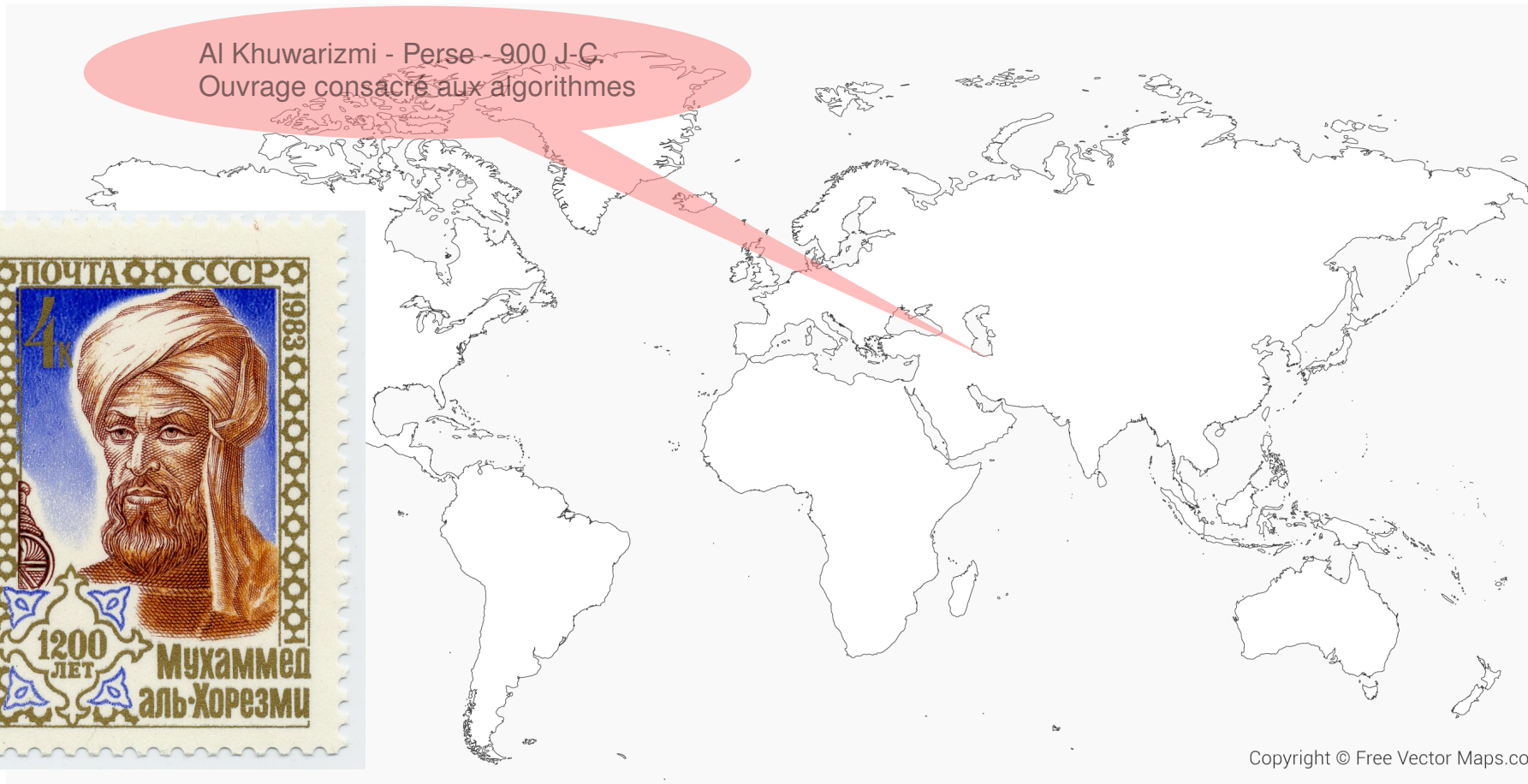
- L_1 Bases de la programmation impérative ;
- L_2 Structures récursives linéaires (et arborescentes) ;
- L_2 Bases de la programmation fonctionnelle ;
- L_2 Logique et Structures Informatiques ;
- L_3 Calcul symbolique ;
- L_3 Compilation ;
- M_1 Calculabilité.

Un peu d'histoire

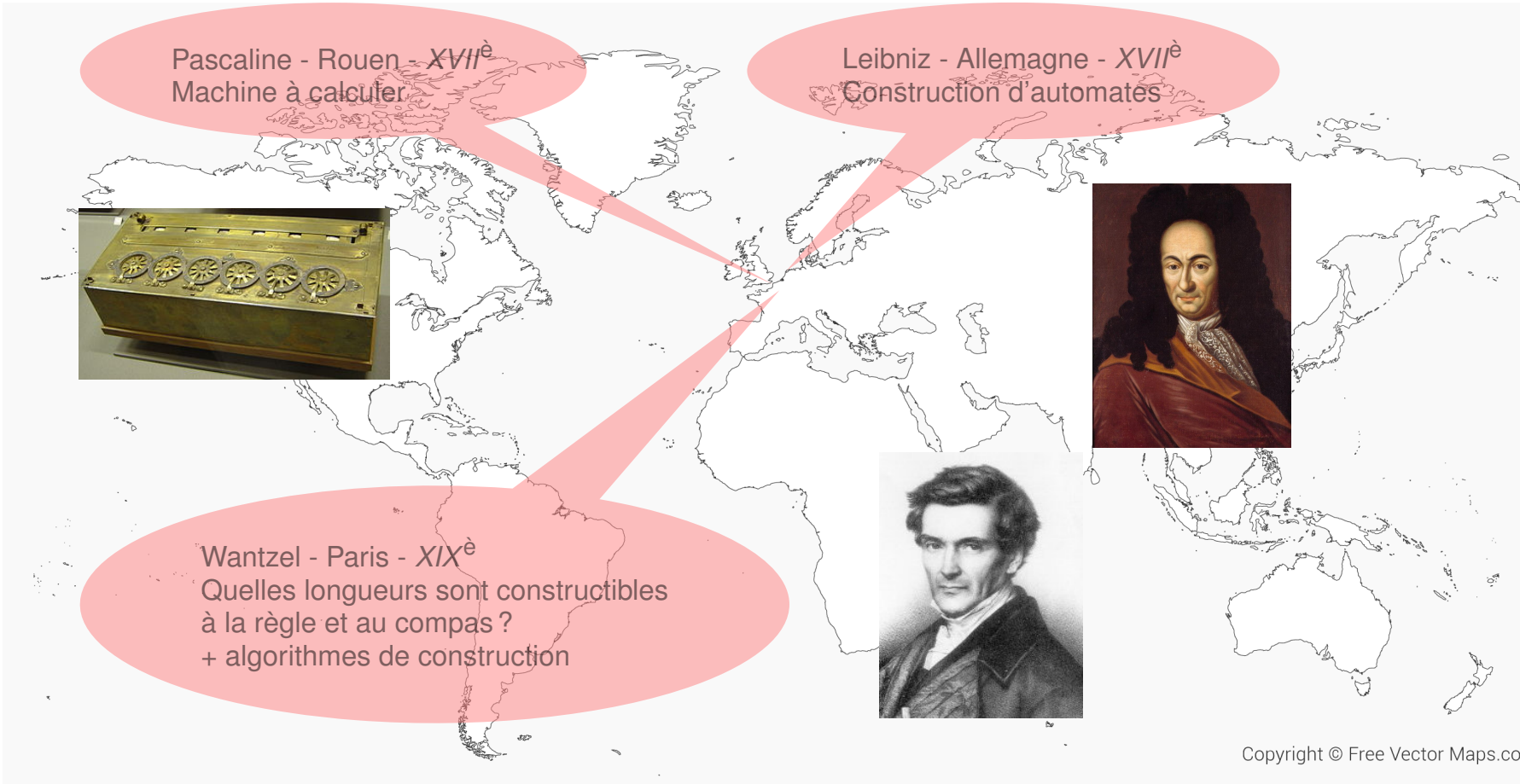
Un peu d'histoire



Un peu d'histoire



Un peu d'histoire



Un peu d'histoire

- Fin XIX^{è} : progrès importants en méta-mathématiques
  - Cantor : développement de la théorie moderne "naïve" des ensembles
  - Zermelo et Fraenkel : axiomatisation de la théorie moderne des ensembles
- Début XX^{è} : programme de Hilbert
  - *Ignorabimus* « Nous ne savons pas et nous ne saurons jamais »*vs* « Nous devons savoir. Nous saurons. »
  - Formalisation des fondements des mathématiques... et d'autres sciences !
  - L'axiomatisation permet la mécanisation de la connaissance
  - 23 problèmes dont :
    1. Hypothèse du continu
    2. Cohérence de la présentation axiomatique de l'arithmétique
    5. Axiomatisation de la physique
    10. Trouver un algorithme déterminant si une équation diophantienne a des solutions

Un peu d'histoire

1928 *Entscheidungsproblem* de Hilbert
Existence d'un algorithme permettant de décider de la véracité d'un énoncé mathématique ?

1929 Gödel montre la complétude du calcul des prédicats du premier ordre

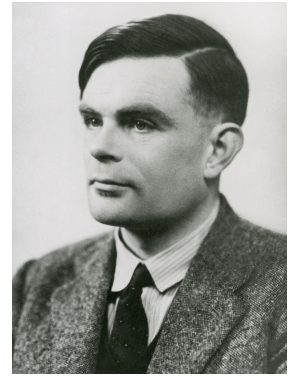
> Si un énoncé est conséquence sémantique d'une théorie, c'est-à-dire s'il est vérifié dans tous les modèles de cette théorie, alors il est conséquence syntaxique de cette théorie : il existe une démonstration formelle dans la syntaxe du calcul des prédicats qui dérive de cet énoncé à partir des axiomes de la théorie en utilisant les règles d'un système de déduction

1931 Théorème d'incomplétude de Gödel

- Toute théorie cohérente capable de formaliser l'arithmétique est incomplète
- La cohérence d'une théorie T (satisfaisant certaines hypothèses raisonnables) n'est pas montrable dans T
  Fin de l'idée de mécanisation, par l'axiomatisation, des mathématiques de Hilbert !

Un peu d'histoire

Church (1930) et Turing (1936)

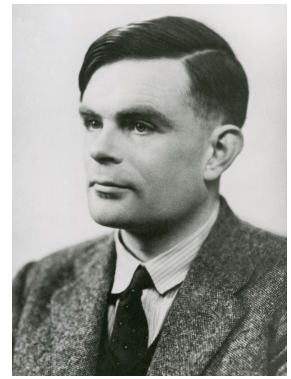


- fonctions définissables en λ-calcul (programmation fonctionnelle)
  = fonctions récursives
  = machines de Turing
- idée de calcul effectif, formalisation de la calculabilité
- machine de Turing = modèle "réaliste" d'ordinateur
- pas encore d'ordinateur construit !
- existence de fonctions non calculables

À la date d'aujourd'hui, les modèles de calcul pour décrire les algorithmes sont encore ceux-ci

Un peu d'histoire

Church (1930) et Turing (1936)



- fonctions définissables en λ-calcul (programmation fonctionnelle)
  = fonctions récursives (programmation fonctionnelle)
  = machines de Turing (programmation impérative)
- idée de calcul effectif, formalisation de la calculabilité
- machine de Turing = modèle "réaliste" d'ordinateur
- pas encore d'ordinateur construit !
- existence de fonctions non calculables

À la date d'aujourd'hui, les modèles de calcul pour décrire les algorithmes sont encore ceux-ci

Programmation impérative : principe

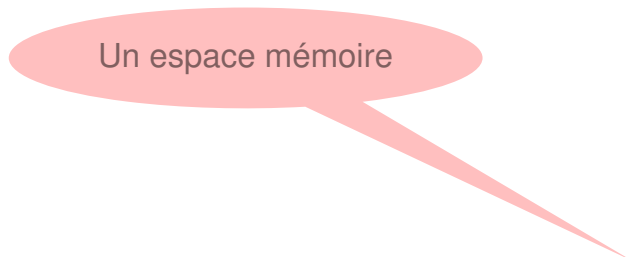
- On modifie des *variables* par des *instructions*
- Les variables sont des espaces mémoire
- Les instructions sont organisées séquentiellement
- La machine se programme par *effets de bords*
  (= modifications de l'état général de la machine (mémoire, vidéo, etc.))

Exemple

```
unsigned long int fact(unsigned int n) {
  unsigned long int r=1UL;
  for (; n>0u; --n)
    r *= n;
  return r;
}
```

Programmation impérative : principe

- On modifie des *variables* par des *instructions*
- Les variables sont des espaces mémoire
- Les instructions sont organisées séquentiellement
- La machine se programme par *effets de bords*
  (= modifications de l'état général de la machine (mémoire, vidéo, etc.))

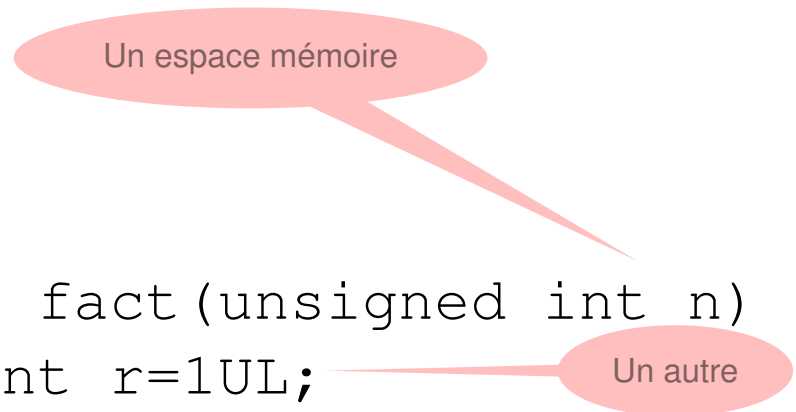


Exemple

```
unsigned long int fact(unsigned int n) {
  unsigned long int r=1UL;
  for (; n>0u; --n)
    r *= n;
  return r;
}
```

Programmation impérative : principe

- On modifie des *variables* par des *instructions*
- Les variables sont des espaces mémoire
- Les instructions sont organisées séquentiellement
- La machine se programme par *effets de bords*
  (= modifications de l'état général de la machine (mémoire, vidéo, etc.))



Exemple

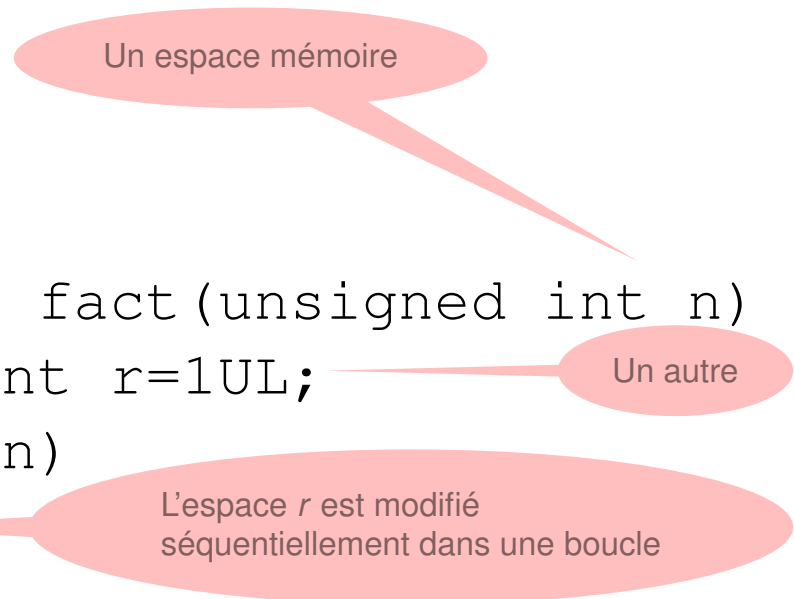
```
unsigned long int fact(unsigned int n) {
  unsigned long int r=1UL;
  for (; n>0u; --n)
    r *= n;
  return r;
}
```

Programmation impérative : principe

- On modifie des *variables* par des *instructions*
- Les variables sont des espaces mémoire
- Les instructions sont organisées séquentiellement
- La machine se programme par *effets de bords*
  (= modifications de l'état général de la machine (mémoire, vidéo, etc.))

Exemple

```
unsigned long int fact(unsigned int n) {
  unsigned long int r=1UL;
  for (; n>0u; --n)
    r *= n;
  return r;
}
```



Programmation impérative : remarques

Les noms des variables sont des noms d'espaces mémoire

$$n = n + 1$$

Impossible ! !

Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs*
  - entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple

```
let rec f = function
  0           -> 1
| n when n>0 -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```

Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs*
  - entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple

f est le nom d'une fonction récursive

```
let rec f = function
  0           -> 1
| n when n>0 -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```

Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs*
  - entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple

f est le nom d'une fonction récursive

qui à 0 associe 1

```
let rec f = function
  0           -> 1
| n when n>0  -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```

Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs*
  - entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple

f est le nom d'une fonction récursive

```
let rec f = function
  0           -> 1
| n when n>0  -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```

qui à 0 associe 1

à toute autre valeur $n > 0$ associe $n * f(n-1)$

Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs*
  - entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple

f est le nom d'une fonction récursive

```
let rec f = function
  0           -> 1
| n when n>0  -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```

qui à 0 associe 1

à toute autre valeur $n > 0$ associe $n * f(n-1)$

et sinon génère une erreur

Programmation fonctionnelle : remarques

L'affectation n'existe plus !

$$n = n + 1$$

Impossible !

Réduction

La *réduction* d'une expression *e* consiste à

- remplacer syntaxiquement dans *e* un nom *n* par sa valeur ;
- appliquer une fonction à son argument pour obtenir une expression.

Exemple

```
let succ = function n -> n+1;;
succ (2*3);;
```

- `succ (2*3)` se réduit en `succ 6`
- `succ 6` se réduit en `(function n -> n+1) 6`
- qui se réduit en `6+1`
- qui se réduit en `7`

Réduction

La *réduction* d'une expression *e* consiste à

- remplacer syntaxiquement dans *e* un nom *n* par sa valeur ;
- appliquer une fonction à son argument pour obtenir une expression.

Exemple

```
let succ = function n -> n+1;;
succ (2*3);;
```

1. `succ (2*3)` se réduit en
   `(function n -> n+1) (2*3)`
2. qui se réduit en `(2*3)+1`
3. qui se réduit en `6+1`
4. qui se réduit en `7`

Réduction

```
let succ = function n -> n+1;;
succ (2*3);;
```

Plusieurs choix d'ordre des réductions possibles :

1. `succ (2*3)` se réduit en `succ 6`
2. `succ 6` se réduit en `(function n -> n+1) 6`
3. qui se réduit en `6+1`
4. qui se réduit en `7`

ou bien

1. `succ (2*3)` se réduit en
   `(function n -> n+1) (2*3)`
2. qui se réduit en `(2*3)+1`
3. qui se réduit en `6+1`
4. qui se réduit en `7`

et d'autres...

Bien sûr, l'ordre choisi des réductions ne doit pas influer sur le résultat (propriété de *confluence* de l'ordre des réductions)

Réduction et preuve

Formellement, une preuve de C à partir d'un ensemble de vérités $V = \{V_1, \ldots, V_n\}$ en utilisant les règles de déduction $D = \{D_1, \ldots, D_k\}$ est exactement une suite de réductions.

Exemple

$V = \{A, B, A \to (B \to C)\}$
$D = \{\text{si } \alpha \text{ et } \alpha \to \beta \text{ alors } \beta\}$

1. $D_1(B, D_1(A, A \to (B \to C)))$ se réduit en $D_1(B, B \to C)$
2. qui se réduit en C.

Une preuve est donc une suite de réductions.

En programmation fonctionnelle, le mécanisme de calcul d'une valeur (à base de réductions) est donc assimilable à la preuve de correction de la valeur.

Programmations impérative *vs* fonctionnelle

Programmation impérative :

- proche de la structure physique de la machine
- permet un contrôle rigoureux des ressources (mémoire, temps, ...)
- gestion complexe de la machine due aux effets de bords

Programmation fonctionnelle (pure) :

- on ne manipule que des définitions mathématiques simples
- le mécanisme d'évaluation est simple
- il fournit directement des preuves de correction des valeurs calculées
- les preuves de correction des programmes sont très simplifiées par l'absence d'effets de bords

Programmations impérative *vs* fonctionnelle

La programmation impérative est utilisée quand un contrôle strict des ressources de la machine est voulu.

Certains systèmes fonctionnels (comme OCaml) fournissent des garanties de sûreté d'évaluation (par le système de typage).

La programmation fonctionnelle est utilisée dans les domaines critiques où

- les preuves de correction des programmes sont importantes
- la sûreté de l'exécution doit être garantie

par exemple l'aéronautique.

Elle est aussi utilisée parce que c'est un style de programmation agréable avec certaines facilités de maintenance.

Tout ce qui peut être programmé dans un des deux paradigmes peut être programmé dans l'autre (attention au coût de la transformation).

Principaux langages fonctionnels

- $\sim$ 1930 : Base théorique λ-calcul (Church)
- 1950 : LISP
  - Scheme
  - Clojure
  - Common LISP
- 1962 : APL
- 1970 : ML
  - Standard ML
  - OCaml
- 1985 : Haskell

Et bien d'autres...

On retrouve aussi des mécanismes de programmation fonctionnelle dans Java 7, par exemple.

Caractéristiques de Objective Caml

OCaml :

- est une implantation du langage Caml
- est développé à l'INRIA
- est un langage fonctionnel impur
  - il est à base de fonctions, qui sont des valeurs comme les autres
  - il possède des mécanismes impératifs et objets
    - pour améliorer l'efficacité de certains programmes
    - nous n'en parlerons pas
- statiquement typé : le typage est effectué avant l'évaluation
- le typage est polymorphe : le système de typage autorise des inconnues de types
- le typage est inféré : c'est OCaml qui le calcule, le programmeur n'a rien à faire (pas de déclaration de type)

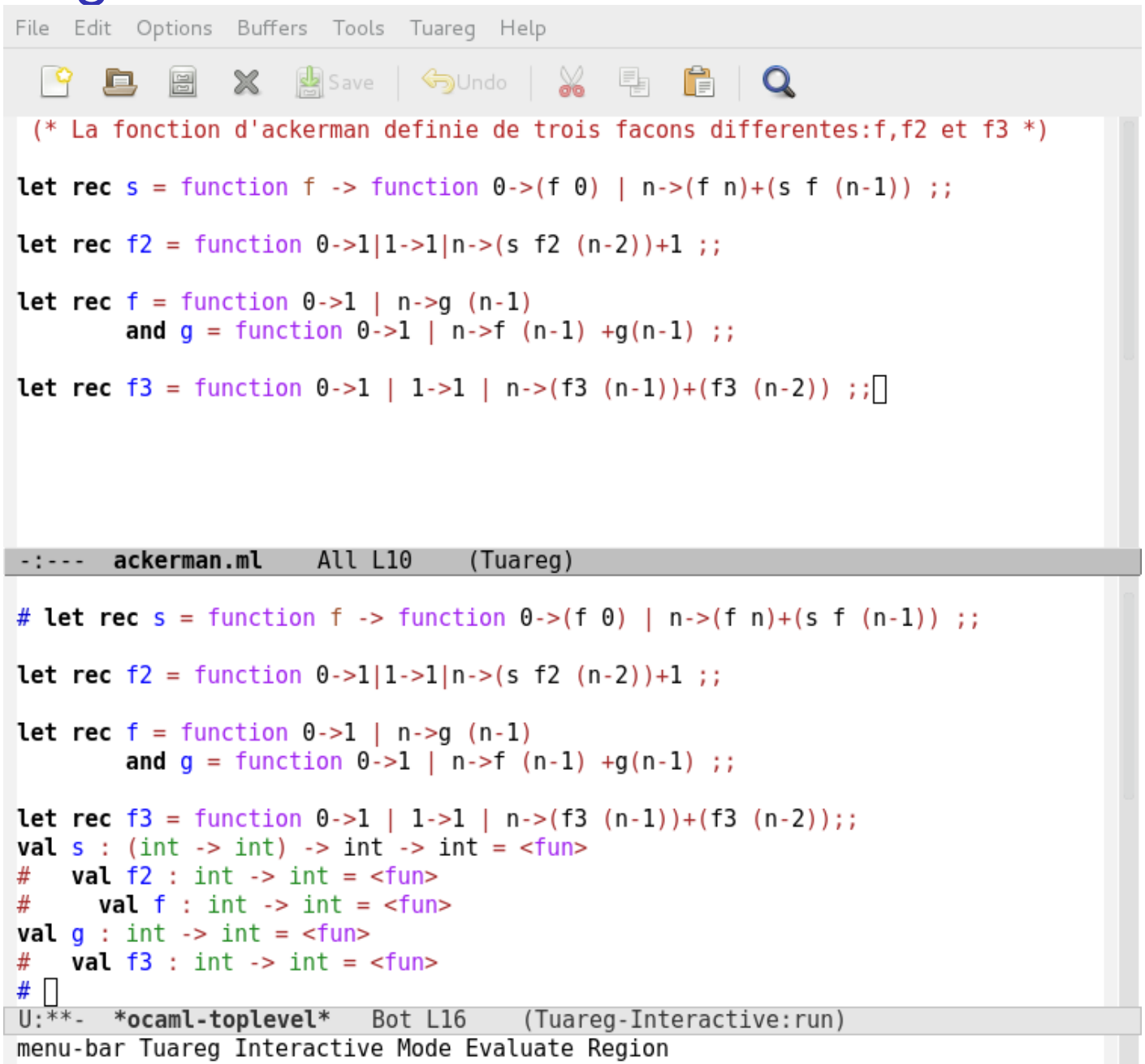
OCaml est un environnement d'*exécution sûr* : son système de typage permet de prévoir (et prouver) les types des grandeurs manipulées pendant l'évaluation, et garantit ainsi l'absence de « surprises » (sur les types) pendant l'évaluation.

Caractéristiques de Objective Caml

OCaml :

- possède un mécanisme de filtrage facilitant les définitions
- possède un mécanisme d'exceptions pour la gestion des erreurs
- contient de nombreuses bibliothèques pour les structures de données classiques, les E/S
- a un environnement de programmation sophistiqué
  - interprète
  - compilateur
  - débogueur
  - analyseur de performances
  - éditeurs/IDE compatibles
    - Emacs : Tuareg
    - Eclipse : OcaIDE
    - ...

Tuareg


File Edit Options Buffers Tools Tuareg Help
Save Undo
(* La fonction d'ackerman definie de trois facons differentes:f,f2 et f3 *)
let rec s = function f -> function 0->(f 0) | n->(f n)+(s f (n-1)) ;;
let rec f2 = function 0->1|1->1|n->(s f2 (n-2))+1 ;;
let rec f = function 0->1 | n->g (n-1)
and g = function 0->1 | n->f (n-1) +g(n-1) ;;
let rec f3 = function 0->1 | 1->1 | n->(f3 (n-1))+(f3 (n-2)) ;;
-:--- ackerman.ml All L10 (Tuareg)
let rec s = function f -> function 0->(f 0) | n->(f n)+(s f (n-1)) ;;
let rec f2 = function 0->1|1->1|n->(s f2 (n-2))+1 ;;
let rec f = function 0->1 | n->g (n-1)
and g = function 0->1 | n->f (n-1) +g(n-1) ;;
let rec f3 = function 0->1 | 1->1 | n->(f3 (n-1))+(f3 (n-2));;
val s : (int -> int) -> int -> int = <fun>
val f2 : int -> int = <fun>
val f : int -> int = <fun>
val g : int -> int = <fun>
val f3 : int -> int = <fun>
#
U:**- *ocaml-toplevel* Bot L16 (Tuareg-Interactive:run)
menu-bar Tuareg Interactive Mode Evaluate Region

Réglages généraux pour B.P.F.

1 Installation d'OCaml

```
sudo apt install ocaml
```

Rappel : Pour la description des fonctions du module de base (*Pervasives*) d'OCaml, ouvrir

https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html.

2 Installation d'OCamlbuild (compilation)

```
sudo apt install ocamlbuild
```

3 Installation d'un environnement java

```
sudo apt install default-jre
```

4 Installation d'Eclipse

1. Télécharger eclipse-installer(https ://www.eclipse.org/downloads/),

Premier exemple

```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

Premier exemple

pour récupérer les lignes antérieures avec les flèches (optionnel, pratique)

```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

Premier exemple

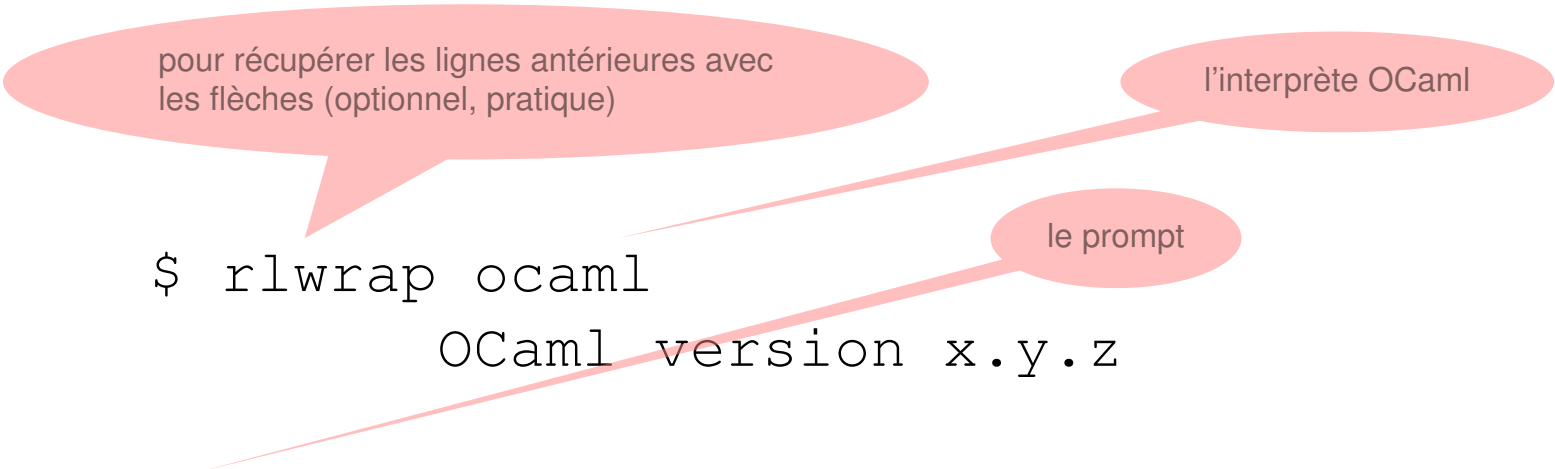
pour récupérer les lignes antérieures avec les flèches (optionnel, pratique)

l'interprète OCaml

```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

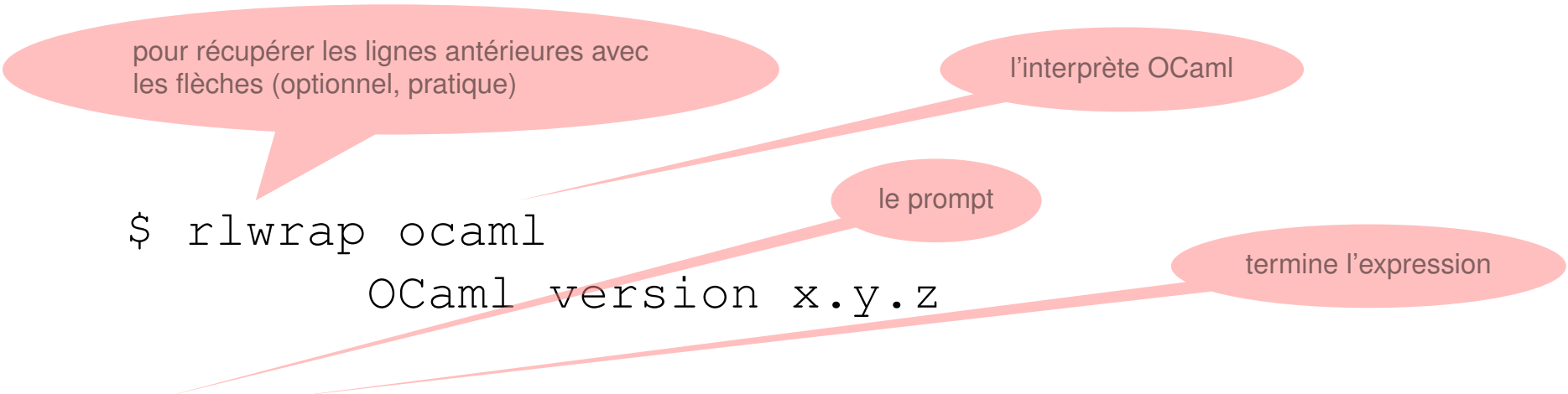
Premier exemple




```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

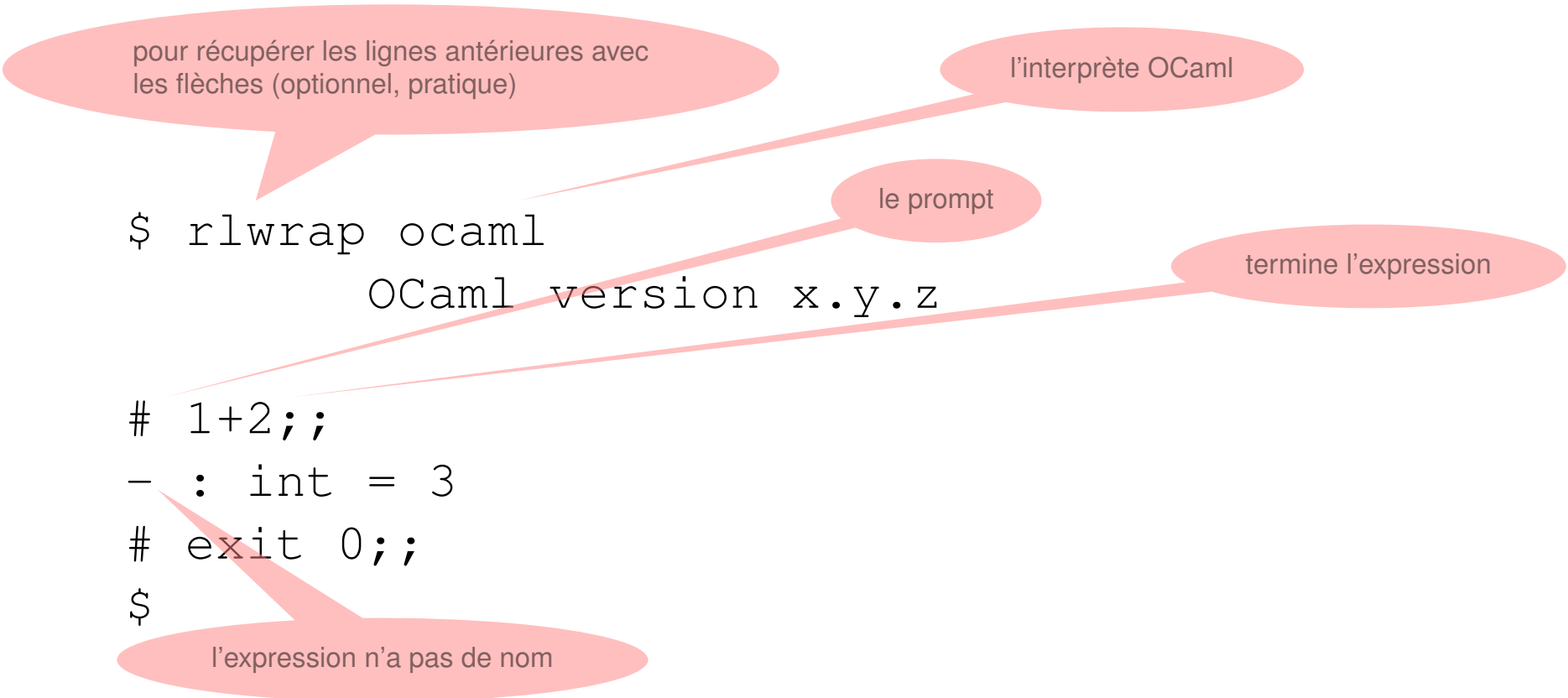
Premier exemple



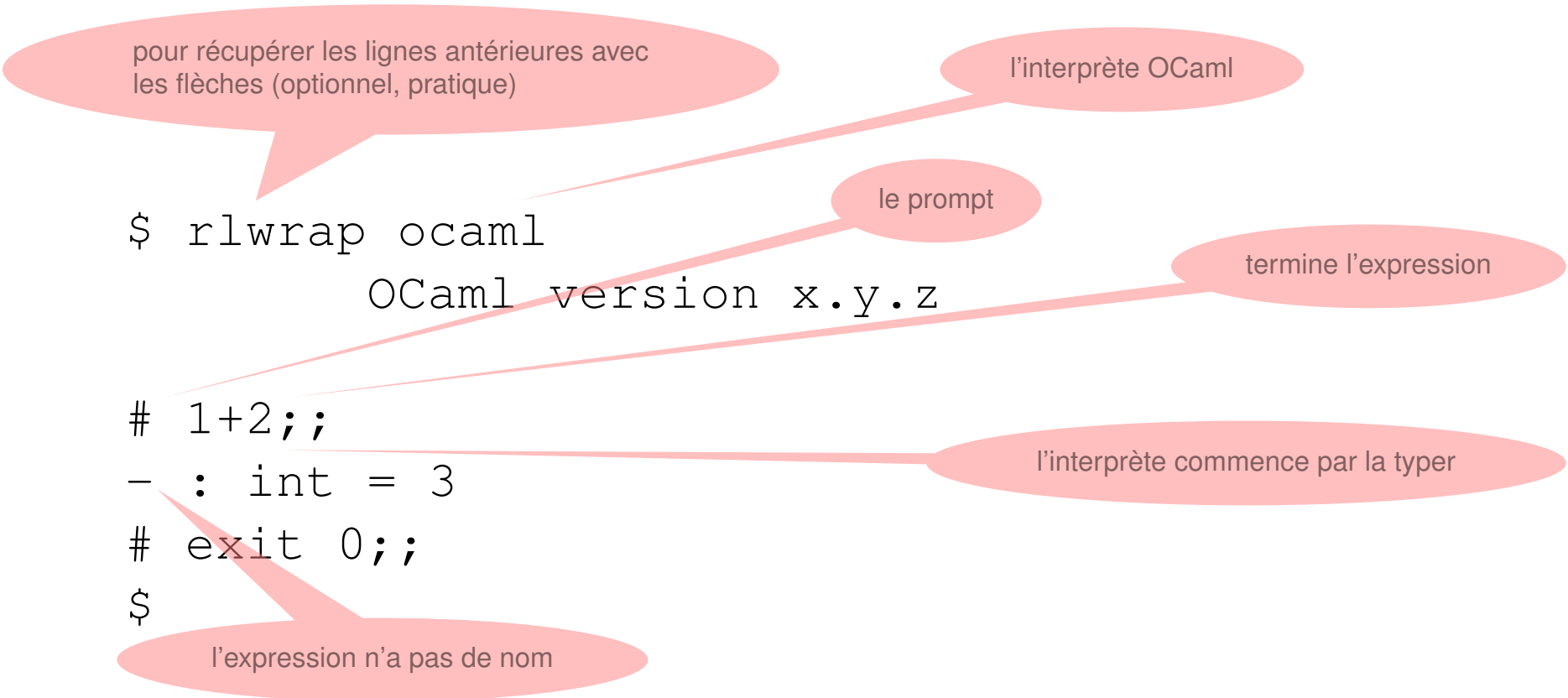

```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

Premier exemple



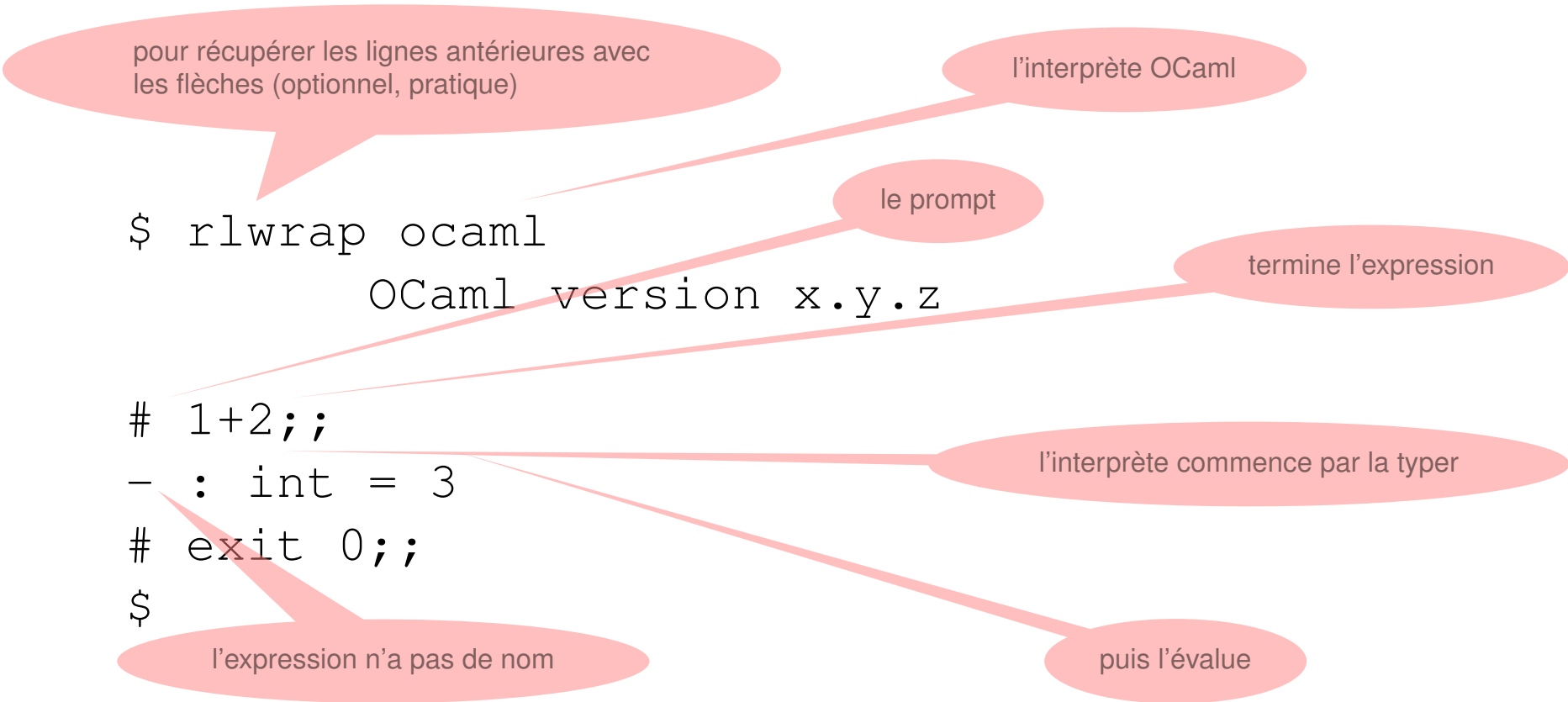
Premier exemple




```
$ rlwrap ocaml
        OCaml version x.y.z

# 1+2;;
- : int = 3
# exit 0;;
$
```

Premier exemple



Un autre exemple

```
# let succ = function x -> x+1;;
val succ : int -> int = <fun>
# succ;;
- : int -> int = <fun>
# succ 4;;
- : int = 5
# let add x y = x+y;;
val add : int -> int -> int = <fun>
# add;;
- : int -> int -> int = <fun>
# add 5 6;;
- : int = 11
# add 5 (succ 6);;
- : int = 12
# add 5;;
- : int -> int = <fun>
# (add 5) (succ 6);;
- : int = 12
```

Encore un autre

```
# let succ x = x+1;;
val succ : int -> int = <fun>
# let succ2 x = succ (succ x);;
val succ2 : int -> int = <fun>
# succ2 5;;
- : int = 7
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply succ 7;;
- : int = 8
# let id f = f;;
val id : 'a -> 'a = <fun>
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

D'autres modes d'exécution : interprète silencieux

```
$ cat exis.ml
let succ x = x+1;;
succ 5;;
print_newline (print_int (succ 5));;
$ ocaml exis.ml
6
$
```

- le fichier est interprété phrase par phrase (phrase=*expression*`;;`) ;
- la valeur et le type de chaque expression ne sont pas affichés ;
- les erreurs continuent d'être affichées sur la sortie erreur standard.

D'autres modes d'exécution : compilation

Les gros programmes peuvent être compilés selon deux modes :

- `ocamlc` produit du byte-code exécutable par une machine virtuelle OCaml : `ocamlrun`

```
$ ocamlc exis.ml
$ ls
a.out  exis.cmi  exis.cmx  exis.ml  exis.o
$ ocamlrun ./a.out
6
$ ./a.out
6
$
```

L'exécutable `a.out` auto-appelle `ocamlrun` (système dépendant)

- `ocamlopt` produit du byte-code natif et optimisé, exécutable directement par la plate-forme

```
$ ocamlopt exis.ml
$ ls
a.out exis.cmi exis.cmx exis.ml exis.o
$ ./a.out
```

Les types de base de OCaml

Sont les

- entiers : `int`
  Codage par complément à deux sur *N*-1 bits (*N*=taille du mot machine)
- réels : `float`
  IEEE 754
- booléens : `bool`
  `true`, `false`
- caractères : `char`
  Codage sur 8 bits instable (ASCII 7 bits puis ? ?)
- chaînes de caractères : `string`
  Codage instable
- `unit`

Les entiers

- Opérateurs arithmétiques : $+$, $-$, $*$, $/$, $\mod$, $-$ (préfixe)
  - non sécurisés : une valeur est quand même fournie en cas de dépassement de capacité
  - la division par 0 génère une exception
- Opérateurs bit à bit logiques : `lor`, `lxor`, `lsl`, `lsr`
- Opérateur bit à bit arithmétique : `asr`
- Opérateurs de comparaisons : $=$, $<$, $>$, $<=$, $>=$

```
# 1;;
- : int = 1
# 1+2;;
- : int = 3
# 4/0;;
Exception: Division_by_zero.
# 4 mod 0;;
Exception: Division_by_zero.
# 99999999999* 9999999999;;
- : int = 3875819909684212737
# 4<5;;
- : bool = true
```

Les réels

- Opérateurs arithmétiques : $+.$, $-.$, $*.$, $/.$, $**$, $-.$ (préfixe)
  - attention aux problèmes de précision
  - sécurisés : valeurs spéciales
    - `infinity` ($+\infty$), `neg_infinity` ($-\infty$)
    - `NaN` (*Not A Number*)
  - la division de $x > 0$ par 0 produit `infinity`
  - la division de $x < 0$ par 0 produit `neg_infinity`
  - la division de 0 par 0 produit NaN
- Opérateurs de comparaison : $=$, $<$, $>$, $<=$, $>=$

Les réels

```
# 1.0;;
- : float = 1.
# 3.14/.0.;;
- : float = infinity
# -1.0/.0.;;
- : float = neg_infinity
# 0./.0.;;
- : float = nan
# -1.0/0.;;
Error: This expression has type float but an expression
         was expected of type int
# 6e34;;
- : float = 6e+34
```

Fonctions sur les réels et les entiers

- Fonctions sur les réels `float` → `float`
  `ceil`, `floor`, `sqrt`, `exp`, `log`, `log10`
- Fonctions trigonométriques sur les réels `float` → `float`
  `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
- Fonctions de conversion
  - `float_of_int` : `int` → `float`
  - `int_of_float` : `float` → `int`

```
# int_of_float 3.14;;
- : int = 3
# int_of_float -3.14;;
Error: This expression has type float -> int
       but an expression was expected of type int
# int_of_float (-3.14);;
- : int = -3
# int_of_float nan;;
- : int = 0
# int_of_float neg_infinity;;
- : int = 0
```

Les caractères

```
# 'a';;
- : char = 'a'
# int_of_char;;
- : char -> int = <fun>
# int_of_char 'a';;
- : int = 97
# char_of_int;;
- : int -> char = <fun>
# char_of_int 97;;
- : char = 'a'
# '\097';;
- : char = 'a';;
# '\n';;
- : char = '\n';;
```

Les caractères

Le module `Char` contient des fonctions qui permettent de manipuler des caractères :

- `Char.lowercase_ascii: char` $\rightarrow$ `char` équivalent minuscule d'un caractère ;
- `Char.uppercase_ascii: char` $\rightarrow$ `char` équivalent majuscule d'un caractère ;
- et d'autres...

L'ordre sur les caractères est l'ordre ASCII-7.

Les chaînes de caractères

```
# "Bonjour monde !";;
- : string = "Bonjour monde !"
# string_of_float;
- : float -> string = <fun>
# (string_of_float 3.14)^" est un nombre formidable !";;
- : string = "3.14 est un nombre formidable !"
# int_of_string;;
- : string -> int = <fun>
# int_of_string "18";;
- : int = 18
# "Bonjour".[5];;
- : char = 'u'
```

Les chaînes de caractères

Il existe aussi des conversions à partir/vers les autres types élémentaires :

- Chaînes $\leftrightarrow$ entiers
  - `string_of_int` : `int` $\rightarrow$ `string`
  - `int_of_string` : `string` $\rightarrow$ `int`
- Chaînes $\leftrightarrow$ réels
  - `string_of_float` : `float` $\rightarrow$ `string`
  - `float_of_string` : `string` $\rightarrow$ `float`
- Chaînes $\leftrightarrow$ booléens
  - `string_of_bool`
  - `bool_of_string`
- `string_of_char` et `char_of_string` n'existent pas :
  - Construire une chaîne à partir d'un caractère : `String.make` : `int` $\rightarrow$ `char` $\rightarrow$ `string`
  - Récupérer un caractère d'une chaîne : opérateur `chaîne.[position]` ou `String.get chaîne position`

Les chaînes de caractères

Le module `String` contient de nombreuses autres fonctions de manipulation de chaînes. Les caractères sont indicés à partir de 0. Une exception est générée en cas d'erreur.

- longueur :
  `String.length : string → int`

```
# String.length "J'aime les poires";;
- : int = 17
```

- sous-chaîne :
  `String.sub : string → int → int → string`

```
# String.sub "J'aime les poires" 7 3;;
- : string = "les"
```

- suppression des espacements au début et à la fin
  `String.trim : string → string`

```
# String.trim "  J'aime les poires   ";;
- : string = "J'aime les poires"
```

Les chaînes de caractères

- despécialisation des caractères spéciaux :
  `String.escaped : string → string`

```
# String.escaped "Il dit \"Bonjour\"\n";;
- : string = "Il dit \\\"Bonjour\\\"\\n"
# String.length "Il dit \"Bonjour\"\n";;
- : int = 17
# String.length
    (String.escaped "Il dit \"Bonjour\"\n");;
- : int = 20
```

- position de la première occurrence d'un caractère :
  `String.index : string → char → int`

```
# String.index "J'aime les poires" 'i';;
- : int = 3
#  String.index "J'aime les poires" 'x';;
Exception: Not_found.
```

- position de la dernière occurrence d'un caractère :
  `String.rindex : string → char → int`
- `String.index_from` et `String.rindex_from` :
  même chose à partir d'une position donnée

Les chaînes de caractères

- `String.contains`, `String.contains_from`, `String.rcontains_from` :
  même chose mais retourne un booléen plutôt qu'une position
- `String.uppercase_ascii`, `String.lowercase_ascii` : `string` → `string`
  mise en majuscule/minuscule

```
# String.uppercase_ascii "J'aime les poires";;
- : string = "J'AIME LES POIRES"
```

- `String.capitalize_ascii`, `String.uncapitalize_ascii` : `string` → `string`
  mise en majuscule/minuscule de la première lettre

```
# String.capitalize_ascii "J'aime les poires";;
- : string = "J'aime les poires"
# String.uncapitalize_ascii "J'aime les poires";;
- : string = "j'aime les poires"
```

Les chaînes de caractères

Complexité des opérations :

- accès à un caractère à partir de sa position : $O(1)$
- recherches : $O(longueur)$
- calcul de la longueur $O(1)$

Ordre sur les chaînes ($<$) :

- il est total, mais inconnu (ça n'est pas nécessairement l'ordre lexicographique)

Les booléens

- Valeurs `true` et `false`
- Opérateurs `not`, `&&`, `||`
- Synonymes (ne pas utiliser)
  - `or` pour `||`
  - `&` pour `&&`
- `false` < `true`
- Évaluation paresseuse de gauche à droite
- Opérateurs de comparaison : polymorphes, mais les opérandes doivent avoir le même type
  - = : égalité structurelle,
  - == : égalité physique (les opérandes occupent le même espace mémoire),
  - <> (négation de =), != (négation de ==),
  - <, >, <=, >=.

Les booléens

Attention :

- = et == retournent la même valeur pour les entiers, les booléens et les caractères et les constructeurs constants ;
- les réels et les chaînes de caractères sont considérées comme des valeurs structurées ;
- préférer = sauf si besoin explicite de ==.

```
# "bonjour"=="bonjour";;
- : bool = false
# "bonjour"="bonjour";;
- : bool = true
# 3.14=3.14;;
- : bool = true
# 3.14==3.14;;
- : bool = false
# 3==3;;
- : bool = true
# 3=3;;
- : bool = true
```

Le type `unit`

- ne contient qu'une seule valeur : `()` ;
- utilisé, par exemple, pour les fonctions à effets de bord dont la valeur de retour peut toujours être négligée.

```
# print_string;;
- : string -> unit = <fun>
# print_string "Bonjour";;
Bonjour- : unit = ()
```

Le produit cartésien

En mathématiques, $A \times B = \{(a, b) : a \in A, b \in B\}$.

En OCaml, $A \times B$ est noté $A * B$ et les couples sont notés a, b. Les parenthèses sont optionnelles mais ont les mets souvent par clarté ou pour lever les ambiguités.

Les fonctions `fst` et `snd` permettent de récupérer les premières et secondes composantes des couples.

L'extension aux n-uplets est naturelle.

Les n-uplets sont lexicographiquement ordonnés par $<$.

Le produit cartésien

```
# 1,true;;
- : int * bool = (1, true)
# 2,true,"bonjour",'a';;
- : int * bool * string * char =
            (2, true, "bonjour", 'a')
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst 2,true;;
Error: This expression has type int but an
            expression was expected of type
          'a * 'b
# fst (2,true);;
- : int = 2
# snd (2,true);;
- : bool = true
```

Le produit cartésien

```
# ((1,'a'),true);;
- : (int * char) * bool = ((1, 'a'), true)
# fst ((1,'a'),true);;
- : int * char = (1, 'a')
# snd (fst ((1,'a'),true));;
- : char = 'a'
# (1,'a',true);;
- : int * char * bool = (1, 'a', true)
# fst (1,'a',true);;
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type 'd * 'e
# (3,true)<(4,false);;
- : bool = true
# (3,true,'b')<(3,true,'c');;
- : bool = true
```

Les listes

En informatique, une *liste* est une structure de données représentant une suite d'éléments, et à accès séquentiel :

- chaque élément est stocké dans un *maillon* ;
- les maillons sont *chaînés* entre eux, dans l'ordre des éléments ;
- pour accéder au i^e élément, il faut passer par tous les j^e avec $j < i$.

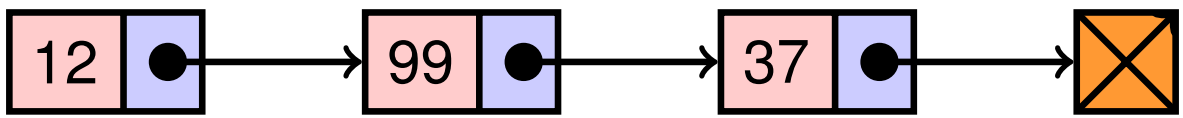


Figure – Une liste simplement chaînée.

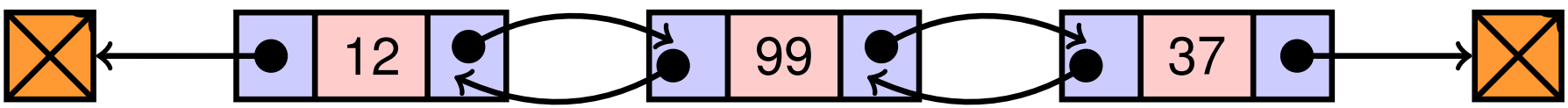


Figure – Une liste doublement chaînée.

Les listes

Définition formelle : une liste à n éléments est soit :

- la liste vide (0 élément : $n = 0$) ;
- un maillon (la *tête* de liste) contenant une valeur, suivi d'une liste (la *queue* de liste) à $n - 1$ éléments.

Exemple d'opérations (liste à n éléments) :

- Accès au i élément : $O(i)$;
- Insertion en tête : $O(1)$;
- Concaténation d'une liste à n éléments avec une liste à m éléments : $O(n)$.

Différences avec les tableaux :

- une liste est une structure de données *dynamique* : sa taille peut varier ;
- Opérations sur un tableau à n éléments :
  - Accès au i élément : $O(1)$;
  - Insertion en tête : $O(n)$;
  - Concaténation d'un tableau à n éléments avec un tableau à m éléments : $O(n + m)$.

Le type `list`

OCaml intègre un type `list` pour les listes.

- la liste vide est notée `[]`
- les listes sont *polymorphes* : les éléments peuvent être de n'importe quel type
- mais elles sont *homogènes* : tous les éléments doivent avoir le même type
- comme les autres valeurs, elles sont non mutables

```
# [];;
- : 'a list = []
# [1;4;2];;
- : int list = [1; 4; 2]
# ['a';'g';'p'];
  ;;
- : char list = ['a'; 'g'; 'p']
# [1;'a'];;
Error: This expression has type char but an
         expression was expected of type int
# [[];[[]]];;
- : 'a list list list = [[]; [[]]]
```

Le type `list`

Construction de nouvelles listes :

`::` (*cons*) permet de construire, à partir d'une liste *l*, une nouvelle liste en lui ajoutant un nouvel élément en tête.

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# 'o'::'c'::'a'::'m'::'l'::[];;
- : char list = ['o'; 'c'; 'a'; 'm'; 'l']
# 1::[];;
- : int list = [1]
# 1::['a'];;
Error: This expression has type char but an
         expression was expected of type int
```

Le *cons* d'un élément à une liste est réalisé en temps constant.
Il est associatif à droite.
En Ocaml les listes sont définies de la manière suivante :

- la liste vide `[]` est une liste ;
- si *l* est une liste, alors $x :: l$ est la liste obtenue par ajout de *x* en tête de *l*, si tous les éléments sont de même type.

Ainsi `::` n'est pas un opérateur, mais un *constructeur*.

Le type `list`

Opérateurs (de construction de nouvelles listes) :

@ est l'opérateur de concaténation de deux listes de même type

```
# [1;2;3]@[4;5];;
- : int list = [1; 2; 3; 4; 5]
# [1;2]@[];;
- : int list = [1; 2]
# [1]@['a'];;
Error: This expression has type char but an
         expression was expected of type int
```

l@l' est réalisé en temps $O(|l|)$ (récursif terminal depuis Ocaml 5.1)

Le type `list`

Fonctions de manipulation du module `List`

- longueur : `List.length: 'a list -> int`

```
# List.length ['a';'b';'c'];;
- : int = 3
```

- tête : `List.hd: 'a list -> 'a`

```
# List.hd ["bonjour";"salut";"hello"];;
- : string = "bonjour"
# List.hd [] ;;
Exception: Failure "hd".
```

- queue : `List.tl: 'a list -> 'a list`

```
# List.tl ["bonjour";"salut";"hello"];;
- : string list = ["salut"; "hello"]
# List.tl ['a'];;
- : char list = []
# List.tl [] ;;
Exception: Failure "tl".
```

Le type `list`

Fonctions de manipulation du module `List`

- nième élément : `List.nth: 'a list -> int -> 'a`

```
# List.nth [3;5;7] 2;;
- : int = 7
# List.nth [3;5;7] 4;;
Exception: Failure "nth".
```

- appartenance : `List.mem: 'a -> 'a list -> bool`

```
# List.mem 5 [3;5;7];;
- : bool = true
# List.mem 12 [3;5;7];;
- : bool = false
```

- renversement : `List.rev: 'a list -> 'a list`

```
# List.rev [3;5;7];;
- : int list = [7; 5; 3]
```

Le type `list`

Fonctions de manipulation du module `List`

- appliquer une fonction à chaque élément :

```
List.map: ('a -> 'b) -> 'a list -> 'b list
```

```
# List.map (function x->x*x) [3;5;7];;
- : int list = [9; 25; 49]
```

- appliquer une fonction sur tous les éléments :

```
List.fold_left: ('a -> 'b -> 'a) -> 'a
                                  -> 'b list -> 'a
```

```
# List.fold_left (fun x y -> x+y) 0 [3;5;7];;
- : int = 15                        (((0+3)+5)+7)
```

- appliquer une fonction sur tous les éléments :

```
List.fold_right: ('a -> 'b -> 'b) -> 'a list
                                   -> 'b -> 'b
```

```
# List.fold_right (fun x y -> x+y) [3;5;7] 0;;
- : int = 15                        (3+(5+(7+0)))
```

- et bien d'autres (voir documentation)

Le type `list`

Fonctions de manipulation du module `List`

- appliquer une fonction à chaque élément :

```
List.map: ('a -> 'b) -> 'a list -> 'b list
```

```
# List.map (function x->log (float_of_int x)) [3;5;7]
- : float list = [1.09..; 1.60..; 1.94..]
```

- appliquer une fonction sur tous les éléments :

```
List.fold_left: ('a -> 'b -> 'a) -> 'a
                                  -> 'b list -> 'a
```

```
# List.fold_left (fun x y -> x-y) 0 [3;5;7];;
- : int = -15                          (((0-3)-5)-7)
```

- appliquer une fonction sur tous les éléments :

```
List.fold_right: ('a -> 'b -> 'b) -> 'a list
                                  -> 'b -> 'b
```

```
# List.fold_right (fun x y -> x-y) [3;5;7] 0;;
- : int = 5                            (3-(5-(7-0)))
```

- et bien d'autres (voir documentation)

Noms symboliques

Donner un nom à une valeur : `let` *nom* `=` *expr* `;;`

```
# x;;
Error: Unbound value x
# let x=2.14+.1.;;
val x : float = 3.14
# let y=4.;;
val y : float = 4.
# x+.y;;
- : float = 7.1400000000000057
# let x=1.;;
val x : float = 1.
# x+.y;;
- : float = 5.
# let x=x+.1.;;  (* Nouvelle utilisation du nom x     *)
val x : float = 2. (* N'est pas une incrémentation ! *)
# x;;
- : float = 2.
```

Noms symboliques

Définir plusieurs noms *simultanément* :

`let` nom_1 `=` $expr_1$ `and ... and` nom_n `=` $expr_n$ `;;`

```
# let x=1 and y=true;;
val x : int = 1
val y : bool = true
# let a='e' and b=a;; (* Les noms sont définis simultané
Error: Unbound value a
# let x=2 and y=x;; (* Attention ! *)
val x : int = 2
val y : int = 1
```

Définir plusieurs noms *séquentiellement* sur la même ligne :

`let` nom_1 `=` $expr_1$ `... let` nom_n `=` $expr_n$ `;;`

```
# let c=2 let d=c;;
val c : int = 2
val d : int = 2
```

Noms symboliques locaux

Définir un nom localement à une expression

`let` *nom* `=` *expr* `in` *expr′* `;;`

```
# let x=1;;
val x : int = 1
# let x=2 in x*x;;
- : int = 4
# x;;
- : int = 1
# (let x=2 in x*x)+x;;
- : int = 5
# (let x=2 and y=3 in x+y)+x;;
- : int = 6
# (let x=2 and y=3 in x+y)+x+(let x=5 in x*x);;
- : int = 31
# let y=2 in let z=y in let y=3 in x+y+z;;
- : int = 6
```

Les conditions

Une expression *e* peut avoir la forme

$$\texttt{if } \textit{condition} \texttt{ then } \textit{expr}_1 \texttt{ else } \textit{expr}_2$$

La valeur de *e* dépend alors de celle de la condition

- si elle est vraie, alors la valeur de l'expression est celle de $\textit{expr}_1$;
- sinon c'est celle de $\textit{expr}_2$

Les types de $\textit{expr}_1$ et $\textit{expr}_2$ doivent être identiques, et définissent le type de l'expression *e*.

```
# 1+(if true then 2 else 3);;
- : int = 3
# 1+(if false then 2 else 3);;
- : int = 4
# 1+(if false then 2 else 3.14);;
Error: This expression has type float but an expression
         was expected of type int
```

Filtrage (*pattern-matching*)

Une expression f peut avoir la forme

$$\texttt{match } e \texttt{ with } v_1 \rightarrow e_1 | \ldots | v_n \rightarrow e_n$$

La valeur de f dépend de celle de e : si c'est v_i alors f vaut e_i.

Le filtrage permet de spécifier des conditions complexes :

Exemple :

```
# if 1+3=0 then "Zéro"
  else if 1+3=1 then "Un"
  else "Autre";;
- : string = "Autre"
```

s'écrit plus simplement

```
# match 1+3 with
  | 0 -> "Zéro"
  | 1 -> "Un"
  | _ -> "Autre";;
- : string = "Autre"
```

Le premier `|` est optionnel. Les défauts de cas sont signalés :

```
# match 1+3 with
  | 0 -> "Zéro"
  | 1 -> "Un";;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
2
Exception: Match_failure ("//toplevel//", 1, 0).
```

Filtrage (*pattern-matching*)

- il est possible de regrouper différents auxquels on associe la même valeur ;
- on peut utiliser _ quand nommer le motif est inutile.

```
# match 'x' with
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y'
  | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y' -> "Voyelle"
  | _ -> "Autre"
  ;;
- : string = "Autre"
```

Filtrage (*pattern-matching*)

Avec le mot-clef `function` il est possible de filtrer implicitement :

```
# let rec fact = function
  0 -> 1
| n when n>0 -> n*(fact (n-1))
| _ -> failwith "Fonction fact non définie pour les enti
;;
# let rec length = function
  [] -> 0
| _::l -> 1+length l
;;
```

Les formes

- `function` $v_1 \to e_1 | \ldots | v_n \to e_n$
- `function` $x \to$ `match` x `with` $v_1 \to e_1 | \ldots | v_n \to e_n$

sont équivalentes.

Filtrage (*pattern-matching*)

Les filtres sont des *valeurs* et non des *expressions* :

```
# let n=3;;
val n : int = 3
# match n with 1+2 -> true | _ -> false;;
Line 1, characters 14-15:
1 | match n with 1+2 -> true | _ -> false;;
                    ^

Error: Syntax error
# match n with 3 -> true | _ -> false;;
- : bool = true
```

Le filtrage permet aussi de *destructurer* l'argument :

```
# match (1,(2,3)) with (x,(y,z)) -> ((x+1,y+1),z+1);;
- : (int * int) * int = ((2, 3), 4)
```

On ne peut pas donner deux fois le même nom à deux parties de motif différentes :

```
# match (1,(2,3)) with (x,(y,x)) -> ((x+1,y+1),z+1);;
Error: Variable x is bound several times
       in this matching
```

Filtrage (*pattern-matching*) sur les listes

Le *cons* `::` (qui n'est pas un opérateur mais un *constructeur*) permet de destructurer suivant le premier élément de la liste.

```
# match [2;4;6] with
    []    -> failwith "Erreur"
  | x::l -> x
;;
- : int = 2
```

Mieux :

```
# match [2;4;6] with
    []    -> failwith "Erreur"
  | x::_ -> x
;;
- : int = 2
```

Filtrage (*pattern-matching*) gardé

Il est possible de poser une condition sur le filtre.

```
# match [(3,"toto");(10,"bidule");(4,"machin")] with
    []         -> failwith "Erreur: liste vide !"
  | (x,_)::l when x=1+List.length l -> l
  | l          -> l
  ;;
- : (int * string) list = [(10, "bidule"); (4, "machin")]
```

Filtrage (*pattern-matching*) gardé

Les conditions ne sont pas évaluées au moment du typage.
Elles peuvent mettre en défaut les tests d'exhaustivité :

```
# let f = function 0 -> 0 | x when true -> x;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
(However, some guarded clause may match this value.)
val f : int -> int = <fun>
```

Il ne faut pas garder tous les cas :

```
# function x when true -> x;;
Warning 25: bad style, all clauses in this
            pattern-matching are guarded.
- : 'a -> 'a = <fun>
```

Nommage par filtrage

```
# let (a,b,c)=(1,true,"Bonjour");;
val a : int = 1
val b : bool = true
val c : string = "Bonjour"
# let (x,y)=(3,4) in x*y;;
- : int = 12
# let (a,_,c)=(1,true,"Bonjour");;
val a : int = 1
val c : string = "Bonjour"
```

Nommage par filtrage

Il est possible de nommer un motif ou une partie décomposée de motif :

```
# match [(3,"toto");(10,"bidule");(4,"machin")] with
    []          -> failwith "Erreur: liste vide !"
  | ((x,_)::l) as l' when x=1+List.length l -> l'
  | x::l              -> l
  ;;
- : (int * string) list = [(3, "toto"); (10, "bidule");
                            (4, "machin")]
```

Les opérateurs

Peuvent être vus comme des fonctions :

```
# ( + );;
- : int -> int -> int = <fun>
# let succ = ( + ) 1;;
val succ : int -> int = <fun>
# succ 4;;
- : int = 5
# ( /. );;
- : float -> float -> float = <fun>
# ( mod );;
- : int -> int -> int = <fun>
# ( lsr );;
- : int -> int -> int = <fun>
# ( :: );; (* :: est un constructeur, pas un opérateur *
Line 1, characters 0-4:
1 | (::);;
    ^^^^
Error: The constructor :: expects 2 argument(s),
       but is applied here to 0 argument(s)
```

Les opérateurs

Il est également possible de définir ses propres opérateurs (les opérateurs « utilisateur » ne doivent contenir ni lettres ni chiffres, et que des symboles).

```
# let ( ++ ) (a,b) (c,d) = (a+c,b+d);;
val ( ++ ) : int * int -> int * int -> int * int = <fun>
# (1,2) ++ (3,4);;
- : int * int = (4, 6)
```

Polymorphisme

Les types de OCaml sont *polymorphes* : ils peuvent contenir des inconnues de types.
Par exemple, les listes sont polymorphes homogènes :

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

Les fonctions `fst` et `snd` retournent respectivement la première et la seconde composante d'un couple de n'« importe quoi » :

```
# let fst = function (x,_) -> x ;;
val fst : 'a * 'b -> 'a = <fun>
# let snd = function (_,y) -> y ;;
val snd : 'a * 'b -> 'b = <fun>
```

Les types de la première et de la seconde composante du couple ne sont pas corrélés.

Polymorphisme

Le système de typage infère le type le plus général possible. La fonction identité retourne son argument, qui peut être de n'importe quel type

```
# let id = function x -> x;;
val id : 'a -> 'a = <fun>
```

Il est possible de donner une valeur à une inconnue de type (forcer un type) :

```
# let id = function (x:int) -> x;;
val id : int -> int = <fun>
```

Définir ses propres types

Il est bien sûr possible de définir ses propres types (types somme, types produit, synonymes, . . .) qui peuvent être paramétrés par des variables de types.

Les types peuvent être récursifs sans qu'il soit nécessaire de l'indiquer.

Les règles de portée de définition de type sont les mêmes que pour les noms symboliques.

Définir ses propres types

Syntaxe quand aucune inconnue de type n'est nécessaire

$$\texttt{type}\ \textit{nom\_type} = \begin{cases} \textit{definition\_type} \\ \textit{type} \end{cases} \quad \texttt{;;}$$

Une définition de type peut être paramétrée par une inconnue de type :

$$\texttt{type}\ {'a}\ \textit{nom\_type} = \begin{cases} \textit{definition\_type} \\ \textit{type} \end{cases} \quad \texttt{;;}$$

ou par plusieurs

$$\texttt{type}\ ({'a_1}, \ldots, {'a_n})\ \textit{nom\_type} = \begin{cases} \textit{definition\_type} \\ \textit{type} \end{cases} \quad \texttt{;;}$$

Plusieurs types peuvent être définis simultanément :

```
type t1 = int list and t2 = char;;
```

La définition simultanée de types est particulièrement utile dans la définition de types mutuellement récursifs.

Types sommes

Un *type somme* est défini par une énumération finie de *constructeurs* qui peuvent prendre des types en argument.
Un nom

- de constructeur commence toujours par une majuscule ;
- de type commence toujours par une minuscule.

```
# type couleur = Trefle | Carreau | Coeur | Pique;;
type couleur = Trefle | Carreau | Coeur | Pique
# Trefle;;
- : couleur = Trefle
# type ma_liste_entiers =
  Vide
| Cons of int*ma_liste_entiers
;;
type ma_liste_entiers =
        Vide | Cons of int*ma_liste_entiers
# let l = Cons (1,Cons(2,Cons(3,Vide)));;
val l : ma_liste_entiers =
    Cons (1, Cons (2, Cons (3, Vide)))
```

Types sommes

Les constructeurs peuvent être utilisés dans le filtrage

```
# let string_of_couleur = function
  Trefle  -> "Trèfle"
| Carreau -> "Carreau"
| Coeur   -> "Coeur"
| Pique   -> "Pique"
;;
val string_of_couleur : couleur -> string = <fun>
string_of_couleur Trefle;;
- : string = "Trèfle"
```

Types sommes

Attention : la notation OCaml pour la définition des constructeurs est malheureuse : le constructeur `Cons` prend deux arguments, et non un couple !

```
# let est_vide = function
    Vide -> true
  | Cons _ -> false
;;
Error: The constructor Cons expects 2 argument(s),
       but is applied here to 1 argument(s)
```

Types sommes

Attention : la notation OCaml pour la définition des constructeurs est malheureuse : le constructeur `Cons` prend deux arguments, et non un couple !
Les ambiguités peuvent être levées par l'emploi de parenthèses :

```
# type ma_liste_entiers =
     Vide
   | Cons of (int*ma_liste_entiers);;
type ma_liste_entiers =
        Vide | Cons of (int * ma_liste_entiers)
# let est_vide = function
    Vide -> true
  | Cons _ -> false
;;
val est_vide : ma_liste_entiers -> bool = <fun>
# est_vide (Cons (1,Cons(2,Cons(3,Vide))));;
- : bool = false
```

Types produits (enregistrements)

Permet de regrouper plusieurs valeurs dans une seule.
Intuition : structure (struct) du C, enregistrement du Pascal.
Syntaxe :

`type` t `= {` nom_1 `:` $type_1$ `;` `...;` nom_n `:` $type_n$ `};;`

Les valeurs se construisent en donnant des valeurs aux champs, entre accolades.
L'accès au champ nom_i d'une valeur v de type t se fait par

- $v.nom_i$
- filtrage

```
# type complex = { re:float; im:float };;
type complex = { re : float; im : float; }
# let add x y = { re=x.re+.y.re; im=x.im+.y.im };;
val add : complex -> complex -> complex = <fun>
# add { re=1.; im=2. } { re=3.; im=4. };;
- : complex = {re = 4.; im = 6.}
```

Types produits (enregistrements)

Attention, les types peuvent se masquer :

```
# add { re=1.; im=2. } { re=3.; im=4. };;
- : complex = {re = 4.; im = 6.}
# type c2 = { re:float; im:float };;
type c2 = { re:float; im:float }
# add { re=1.; im=2. } { re=3.; im=4. };;
          ^
Error: This expression has type c2 but an expression
         was expected of type complex
```

Types produits (enregistrements)

Les enregistrements peuvent être dé-structurés dans un filtrage.

Lors d'un filtrage, il peut ne pas être nécessaire de nommer tous les champs mais le faire permet de lever des ambiguïtés.

```
# let im = function { im=y } -> y;; (* let im {im} = im *)
val im : c2 -> float = <fun>
# type t3 = { im:float };;
type t3 = { im:float }
# let im2 = function { im=y } -> y;;
val im2 : t3 -> float = <fun>
# let im3 = function { re=_; im=y } -> y;;
val im3 : c2 -> float = <fun>
```

Types produits (enregistrements)

Il est possible de définir une valeur à partir d'une autre :

```
# type personne =
    { nom:string; prenom:string; age:int; };;
type personne =
    { nom : string; prenom : string; age : int; }
# let p = { nom="Durand"; prenom="Jean"; age=18 };;
val p : personne =
    { nom = "Durand"; prenom = "Jean"; age = 18 }
# let p' = { p with prenom="Jacques"; age=20 };;
val p' : personne =
    { nom = "Durand"; prenom = "Jacques"; age = 20 }
# p;;
- : personne =
    { nom = "Durand"; prenom = "Jean"; age = 18 }
```

Types produits (enregistrements)

Comme les *n*-uplets, les enregistrements sont lexicographiquement ordonnés par $<$:

```
# type mois = Janvier | Fevrier | Mars | Avril | Mai | Juin
              | Juillet | Aout | Septembre | Octobre | Novembre | Decembre;;
....
# type date = { annee:int; mois:mois; jour:int };;
type date = { annee : int; mois : mois; jour : int; }
# { annee=2000; mois=Avril; jour=10 } < { annee=2000; mois=Avril; jour=10 };;
- : bool = false
# { annee=2000; mois=Avril; jour=10 } < { annee=2001; mois=Avril; jour=10 };;
- : bool = true
# { annee=2000; mois=Avril; jour=10 } < { annee=2000; mois=Mai; jour=10 };;
- : bool = true
# { annee=2000; mois=Avril; jour=10 } < { annee=2000; mois=Avril; jour=10 };;
- : bool = false
# { annee=2000; mois=Avril; jour=10 } < { annee=2001; mois=Avril; jour=10 };;
- : bool = true
# { annee=2000; mois=Avril; jour=10 } < { annee=2000; mois=Mai; jour=10 };;
- : bool = true
# { annee=2000; mois=Avril; jour=10 } < { annee=2000; mois=Avril; jour=11 };;
- : bool = true
```

Types polymorphes

Les définitions de types peuvent être paramétrées par des variables de types :

```
# type 'a ma_liste =
    Vide
  | Cons of 'a*'a ma_liste
;;
type 'a ma_liste = Vide | Cons of 'a*'a ma_liste
# Cons (1,Vide);;
- : int ma_liste = Cons (1, Vide)
# Cons (1,Cons('a',Vide));;
                ^^^
Error: This expression has type char but an
       expression was expected of type int
# type ma_liste_entiers = int ma_liste;;
type ma_liste_entiers = int ma_liste
# Cons (1,Vide);;
- : int ma_liste = Cons (1, Vide)
# type ('a,'b,'c) t = { x:'a; y:'b; z:'c };;
type ('a, 'b, 'c) t = { x : 'a; y : 'b; z : 'c; }
```

Les exceptions

Mécanisme de gestion des erreurs présent dans de nombreux langages de programmation (C++, Java, OCaml, ...)

En OCaml, principalement utilisé pour les fonctions partielles.

```
# 1/0;;
Exception: Division_by_zero.

# match [] with
  []    -> failwith "Pas de tête de liste"
| x::l -> x
;;
Exception: Failure "Pas de tête de liste".
```

Les exceptions

Les exceptions ont le type `exn`.

`exn` est un type somme spécial car extensible (il est possible de lui ajouter de nouveaux constructeurs).

`Failure` est un constructeur pré-défini dont le paramètre est un message d'erreur (chaîne de caractères). Il en existe d'autres.

```
# Failure "toto";;
- : exn = Failure "toto"
```

`failwith: string -> 'a` est une fonction prédéfinie : `failwith s` génère l'exception `Failure s`

Les exceptions sont générées par `raise: exn -> 'a`
Remarques :

- `failwith s` est équivalent à `raise (Failure s)`
- le type de retour de `raise` est indépendant du type de son argument !

Les exceptions

Ajout d'un nouveau constructeur à `exn` :

- `exception` Nom`;;`
- `exception` Nom `of` type`;;`

On rappelle que `exn` est le seul type somme extensible.

```
# exception MonIntException of int;;
exception MonIntException of int
# let fact = function
    n when n<0 -> raise (MonIntException n)
  | _ -> let rec fact_rec acc = function
             0 -> acc
           | n -> fact_rec (acc*n) (n-1)
         in fact_rec 1 n
;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
# fact (-3);;
Exception: MonIntException (-3).
```

Les exceptions

Gestion par un mécanisme de la forme try/catch :

$$\begin{array}{l}\texttt{try}\ \textit{expr}\ \texttt{with}\\ \quad e_1 \rightarrow \textit{expr}_1\\ \mid \ldots\\ \mid e_n \rightarrow \textit{expr}_n\end{array}$$

La valeur de l'expression formée par le `try` est

- celle de *expr* si l'évaluation de *expr* ne génère pas d'exception
- celle de $\textit{expr}_i$ si l'évaluation de *expr* a généré l'exception e_i
- `raise e` si l'évaluation de *expr* a généré l'exception *e* qui n'est aucune des e_i

Remarque : implicitement,

- la liste des exceptions est exhaustive
- ou, vu d'une manière différente, le dernier motif est $\mid e \rightarrow$ `raise` e

Les exceptions

```
# exception IllegalArgumentException of int;;
exception IllegalArgumentException of int
# let fact = function
    n when n<0 -> raise (IllegalArgumentException n)
  | _ -> let rec fact_rec acc = function
              0 -> acc
            | n -> fact_rec (acc*n) (n-1)
         in fact_rec 1 n
;;
val fact : int -> int = <fun>
# let fact_string n =
  try string_of_int (fact n) with
    IllegalArgumentException _ -> "oh my god !"
;;
val fact_string : int -> string = <fun>
# fact_string 5;;
- : string = "120"
# fact_string (-1);;
- : string = "oh my god !"
```

Le type `option`

```
type 'a option = None | Some of 'a
```

Permet de définir des fonctions partielles :

```
let fact n =
  if n<0 then None
  else let rec fact' aux = function
                    0 -> aux
                  | n -> fact' (n*aux) (n-1)
       in Some (fact' 1 n)
;;
val fact : int -> int option = <fun>
# fact 5;;
- : int option = Some 120
# fact (-4);;
- : int option = None
```

Le type `option`

Permet de définir des fonctions à arguments « optionnels » :

```
# let max x y = function
    None   -> if x<y then y else x
  | Some z -> let m = if x<y then y else x
              in if m<z then z else m
  ;;
val max : 'a -> 'a -> 'a option -> 'a = <fun>
# max 4 3 (Some 6);;
- : int = 6
# max 4 3 None;;
- : int = 4
```

Les fonctions en mathématiques

- Ce sont des objets associant de manière unique, à chaque élément d'un ensemble A, un élément d'un ensemble B
- A s'appelle le *domaine (de définition)* de la fonction
- B est son *codomaine*
- On note $f : A \to B$ une fonction de (domaine) A dans (le codomaine) B
- La fonction est *partielle* si elle n'est pas capable d'associer un élément de B à tout élément de A
- On note B^A l'ensemble des fonctions de A dans B (pensez cardinalité !)

Les fonctions en mathématiques

Exemple

- $\log : \mathbb{R} \to \mathbb{R}$ est partiellement définie sur son domaine.
- $\log : \mathbb{R} - \{0\} \to \mathbb{R}$ est complètement définie sur son domaine.
- Soit $n \in \mathbb{N}$. On note $\mathbb{R}^n$ l'ensemble des suites de réels de longueur n et $\mathbb{R}^* = \cup_{n \in \mathbb{N}} \mathbb{R}^n$. On peut définir une fonction taille : $\mathbb{R}^* \to \mathbb{N}$ par

$$\text{taille}(s) = n \text{ si et seulement si } s \in \mathbb{R}^n$$

Fonctions à plusieurs arguments

Mathématiquement n'existent pas mais peuvent être vues comme :

- des fonctions prenant en argument un *n*-uplet
  $*: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- des fonctions retournant des fonctions à $n-1$ arguments
  $*: \mathbb{N} \to (\mathbb{N}^{\mathbb{N}})$

Une fonction de la première forme peut toujours être transformée en une fonction de la seconde forme (*curryfication*) :

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

La transformation dans l'autre sens (*dé-curryfication*) est aussi toujours possible.

Fonctions à plusieurs arguments

En programmation fonctionnelle

- la convention adoptée est la forme curryfiée et donc une fonction prenant en argument un couple a bien un seul argument !
- l'ensemble des fonctions de A dans B est noté par le type des fonctions : $A \to B$
  Par exemple, l'ensemble des fonctions prenant en argument un entier, puis un booléen et retournant un réel est noté
  $$\texttt{int} \to (\texttt{bool} \to \texttt{float})$$
- dans les types, l'opérateur $\to$ est associatif à droite
  $\texttt{int} \to (\texttt{bool} \to \texttt{float})$ s'écrit plus simplement
  $\texttt{int} \to \texttt{bool} \to \texttt{float}$

Fonctions à plusieurs arguments

En informatique, l'*arité* d'une fonction est son nombre d'arguments

```
# let somme x y = x+y;;
val somme : int -> int -> int = <fun>
# let somme_5 = somme 5;;
val somme_5 : int -> int = <fun>
# somme_5 6;;
- : int = 11
# String.sub;;
- : string -> int -> int -> string = <fun>
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a)
                         -> 'c -> 'b = <fun>
```

Fonctions à plusieurs arguments

Deux manières équivalentes d'écrire la fonction `somme` avec deux arguments :

- ```
  function x -> function y -> x+y;;
  - : int -> int -> int = <fun>
  ```
- ```
  fun x y -> x+y;;
  ```
  ou, avec un nom :
  ```
  let somme x y = x+y;;
  - : int -> int -> int = <fun>
  ```

à différencier de cette définition à un seul argument :

```
function (x,y) -> x+y;;
- : int*int -> int = <fun>
```

Fonctions

Une définition de fonction est composée

- d'un nom de paramètre ;
- d'une flèche ;
- d'un corps (expression qui dépend (ou non) du paramètre).

```
# function x -> x+1;;
- : int -> int = <fun>
# (function x -> x+1) 5;;
- : int = 6
# function x -> function y -> x+y;;
- : int -> int -> int = <fun>
# (function x -> function y -> x+y) 1 2;;
- : int = 3
# (function x -> function y -> x+y) 1;;
- : int -> int = <fun>
# ((function x -> function y -> x+y) 1) 2;;
- : int = 3
```

Fonctions

Il est plus pratique de mettre un nom à une fonction

```
let succ = function x -> x+1;;
let add = function x -> function y -> x+y;;
```

`function` n'est valable que pour les fonctions à un seul paramètre. Pour plusieurs on peut utiliser `fun` :

```
let succ = fun x -> x+1;;
let add = fun x y -> x+y;;
```

ou plus directement

```
let succ x = x+1;;
let add x y = x+y;;
```

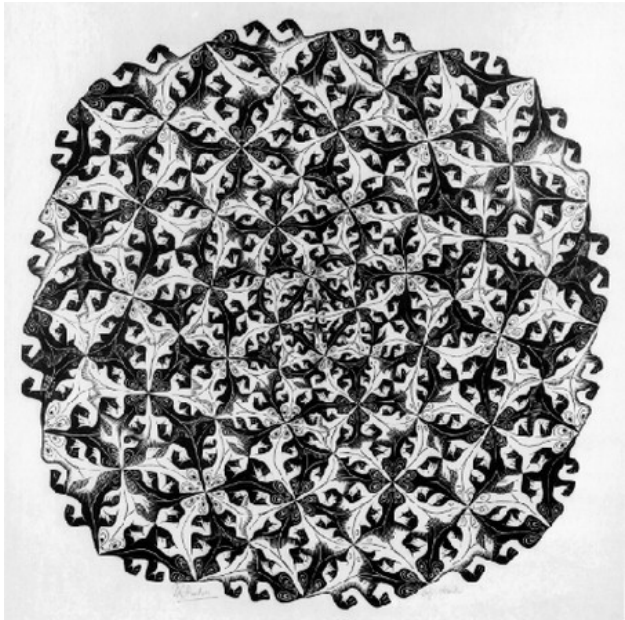
L'application d'une fonction est l'opérateur le plus prioritaire :

```
# succ 2*3;;
- : int = 9
```

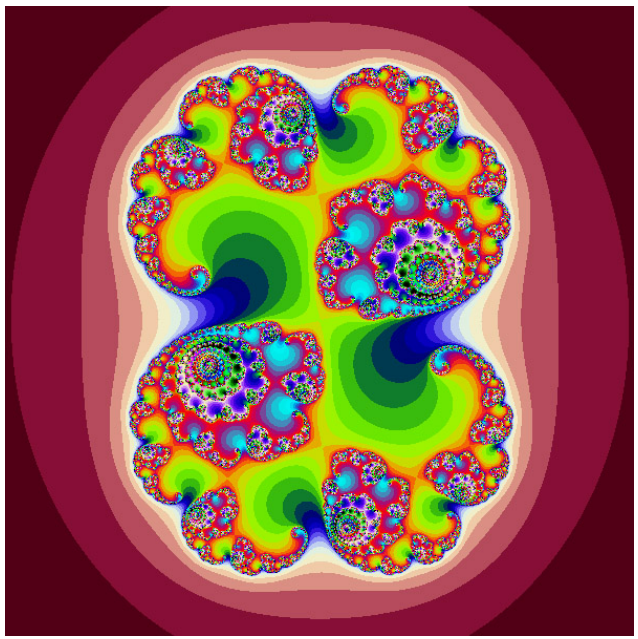
Rappels sur la récursivité

Un objet est *récursif* s'il est défini en appliquant la même règle un nombre indéfini de fois.

Publicité, 1900



Escher, gravure sur bois, 1959



Une courbe de Julia

Rappels sur la récursivité

Un objet est *récursif* s'il est défini en appliquant la même règle un nombre indéfini de fois.

Exemple

Suite des nombres factoriels : $(F_n)_{n\in\mathbb{N}}$, $F_n = n!$

$$F_n = \begin{cases} 1 & \text{si } n = 0 \\ n * F_{n-1} & \text{si } n > 0 \end{cases}$$

Exemple

Suite de Fibonacci :

$$F_n = \begin{cases} 1 & \text{si } n < 2 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

Rappels sur la récursivité

Pour qu'une définition récursive définisse effectivement un objet, il est nécessaire que les paramètres de la définition tendent vers un ou plusieurs *cas de base* (définition *bien fondée*).

Exemple

Suite des nombres factoriels : $(F_n)_{n\in\mathbb{N}}$, $F_n = n!$

$$F_n = \begin{cases} 1 & \text{si } n = 0 \\ n * F_{n-1} & \text{si } n > 0 \end{cases}$$

Pour tout $n \in \mathbb{N}$, la suite n, $n-1$, $n-2$, ..., décroit vers 0

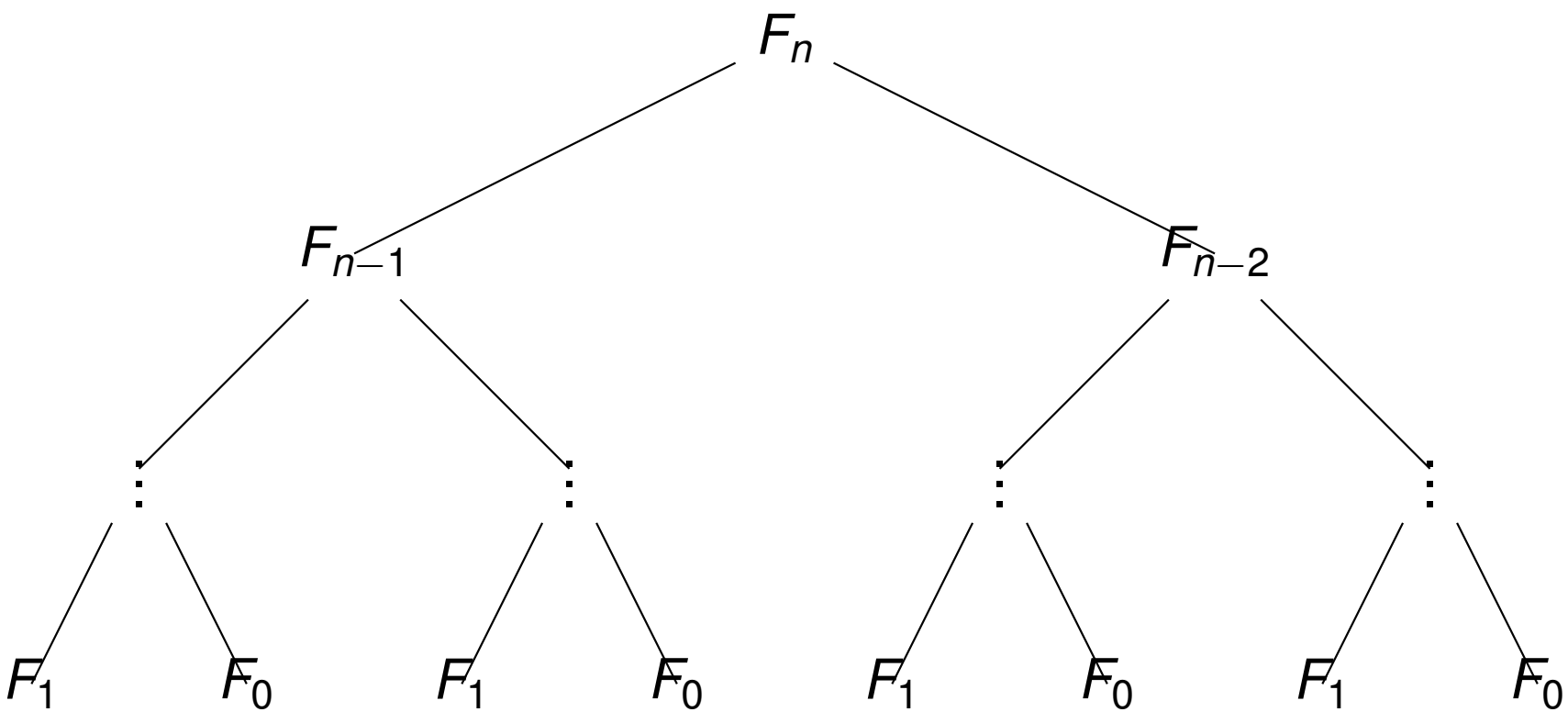
- Le bon fondement d'un calcul récursif garantit la terminaison du calcul.
- Le bon fondement d'une définition récursive lui garantit un sens (sémantique).

Rappels sur la récursivité

Exemple

Suite de Fibonacci :

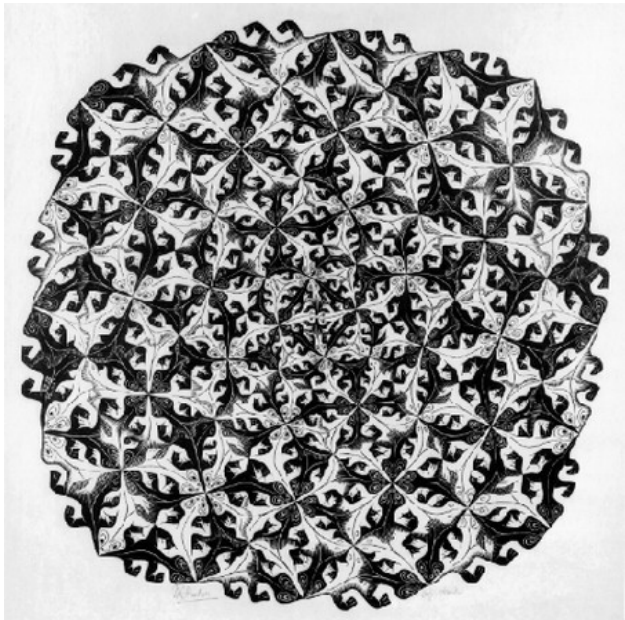
$$F_n = \begin{cases} 1 & \text{si } n < 2 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$



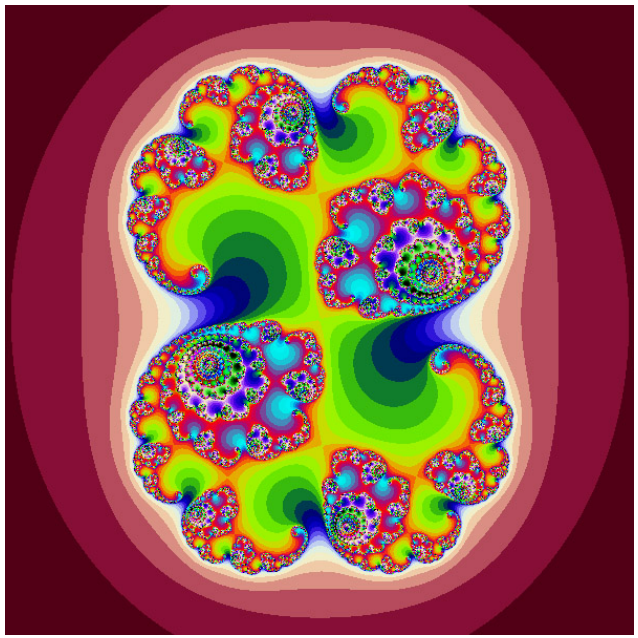
Rappels sur la récursivité

Pour qu'une définition récursive définisse effectivement un objet, il est nécessaire que les paramètres de la définition tendent vers un ou plusieurs *cas de base* (définition *bien fondée*).

Publicité, 1900



Escher, gravure sur bois, 1959



Une courbe de Julia

Rappels sur la récursivité

Pour qu'une définition récursive définisse effectivement un objet, il est nécessaire que les paramètres de la définition tendent vers un ou plusieurs *cas de base* (définition *bien fondée*).

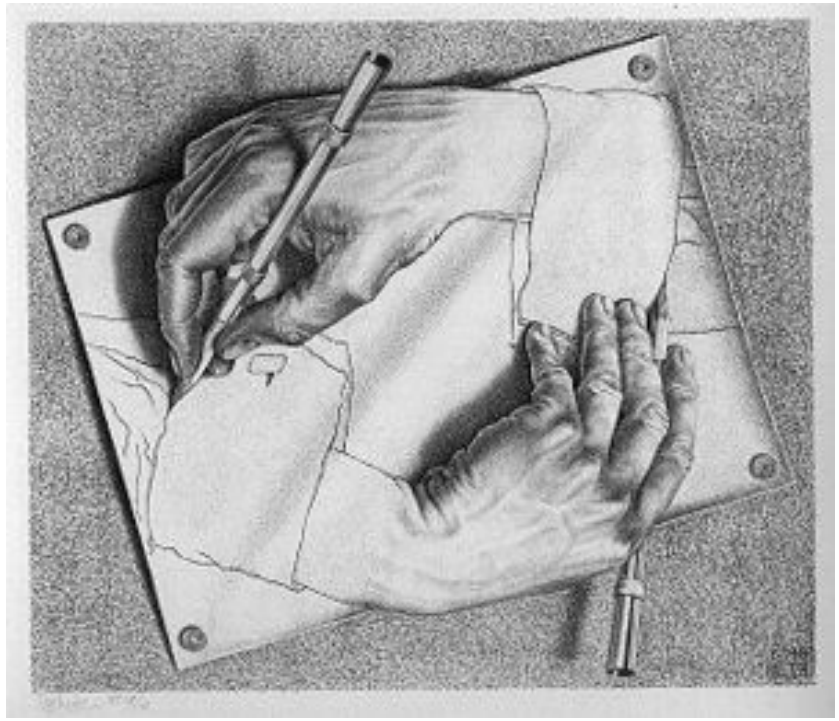


Figure – Lithographie d'Escher. Ici, l'absence de cas de base génère une circularité.

Rappels sur la récursivité

En mathématiques, une suite finie $s_0, \ldots, s_{n-1}$ de n éléments de T est soit :

- la suite vide (de taille 0, $n = 0$) ;
- une suite non vide, composée :
  - d'un élément s_0 de T
  - d'une suite $s_1, \ldots, s_{n-1}$ de $n - 1$ éléments de T

En programmation fonctionnelle, une *liste* d'éléments de T est une suite finie d'éléments de T

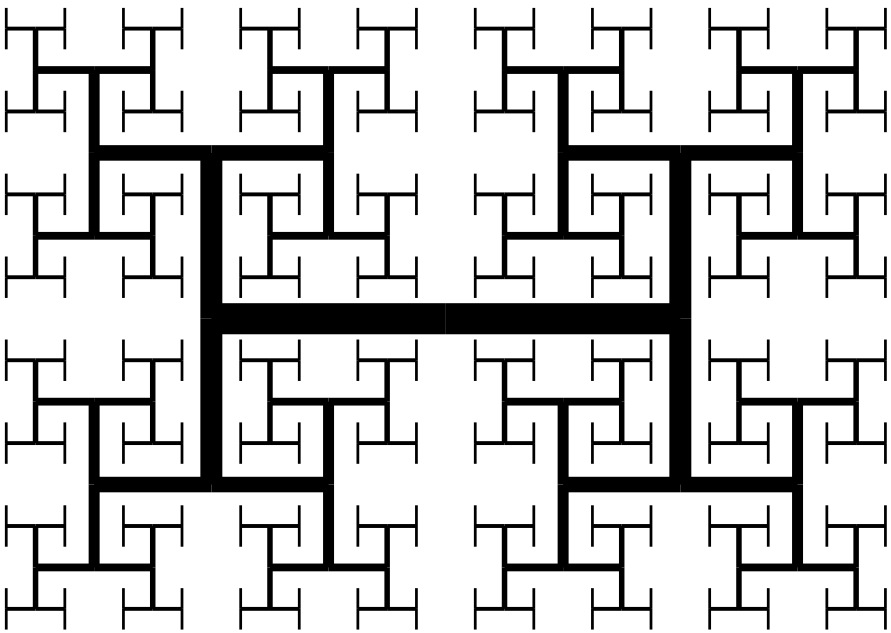
- le premier élément est appelé *tête* (*head*) de liste
- le reste est appelé *queue* (*tail*)

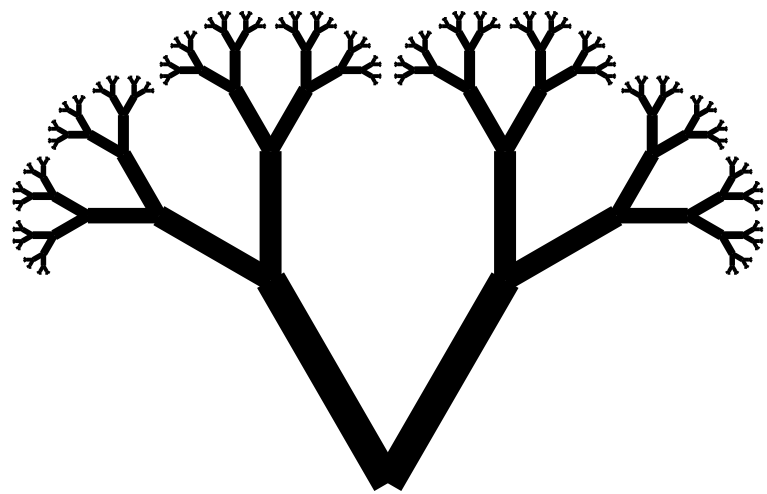
Rappels sur la récursivité

En informatique, un *arbre* est soit

- l'arbre vide
- un nœud contenant un ensemble d'arbres (appelés *fils*).

L'ensemble d'arbres contenu dans un nœud peut être ordonné.
Un nœud sans fils est une *feuille*.





Les fonctions récursives

Les fonctions peuvent être récursives :

```
# let rec fact = if n=0 then 1 else n*(fact (n-1));;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

Les fonctions mutuellement récursives se définissent simultanément :

```
# let rec est_pair n = if n=0 then true
                       else est_impair (n-1)
      and est_impair n = if n=0 then false
                         else est_pair (n-1);;
val est_pair : int -> bool = <fun>
val est_impair : int -> bool = <fun>
```

Les fonctions récursives

Exemple

Exponentiation « égyptienne » :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let rec puiss a = function
    0 -> 1
  | n -> if n mod 2=0 then puissPaire a n
                      else puissImpaire a n
  and puissPaire a n = (puiss a (n/2))*(puiss a (n/2))
  and puissImpaire a n =
                       a*(puiss a (n/2))*(puiss a (n/2));;
val puiss : int -> int -> int = <fun>
val puissPaire : int -> int -> int = <fun>
val puissImpaire : int -> int -> int = <fun>
```

Les fonctions récursives

Exemple

Exponentiation « égyptienne » :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

Calculs dupliqués = complexité incorrecte !

```
# let rec puiss a = function
    0 -> 1
  | n -> if n mod 2=0 then puissPaire a n
                      else puissImpaire a n
  and puissPaire a n = (puiss a (n/2))*(puiss a (n/2))
  and puissImpaire a n =
                       a*(puiss a (n/2))*(puiss a (n/2));;
val puiss : int -> int -> int = <fun>
val puissPaire : int -> int -> int = <fun>
val puissImpaire : int -> int -> int = <fun>
```

Les fonctions récursives

Exemple

Exponentiation « égyptienne » :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let rec puiss a n =
      if n=0 then 1
      else if n mod 2=0 then puissPaire a n
           else puissImpaire a n
  and puissPaire a n =
        (let x = (puiss a (n/2)) in x*x)
  and puissImpaire a n =
        let x = (puiss a (n/2)) in a*x*x;;
val puiss : int -> int -> int = <fun>
val puissPaire : int -> int -> int = <fun>
val puissImpaire : int -> int -> int = <fun>
```

Les fonctions récursives

Exemple

Exponentiation « égyptienne » :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let puiss a n =
    let rec puiss' n =
        if n=0 then 1
        else if n mod 2=0 then puissPaire n
             else puissImpaire n
    and puissPaire n =
         (let x = (puiss' (n/2)) in x*x)
    and puissImpaire n =
         (let x = (puiss' (n/2)) in a*x*x)
  in puiss' n;;
val puiss : int -> int -> int = <fun>
```

Les fonctions récursives

Exemple

Exponentiation « égyptienne » :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let puiss a n =
    let rec puiss' = function
        0                    -> 1
      | n when n mod 2=0 -> puissPaire n
      | n                    -> puissImpaire n
    and puissPaire n =
         (let x = (puiss' (n/2)) in x*x)
    and puissImpaire n =
         (let x = (puiss' (n/2)) in a*x*x)
    in if n<0 then failwith "puiss a n: n<0 !"
       else puiss' n;;
val puiss : int -> int -> int = <fun>
```

Programmation récursive efficace

À chaque appel récursif, il faut empiler l'ensemble des paramètres. Dans certains cas (récursivité terminale), les appels récursifs peuvent être transformés en une boucle.

Définition

Une fonction est *récursive terminale* (en anglais : *tail-recursive*) si elle est récursive et si l'appel récursif est le dernier terme à évaluer.

Programmation récursive efficace

Exemple

```
let rec fact n = if n=0 then 1 else n*(fact (n-1));;
let fact' n =
   let rec f acc n = if n=0 then acc
                     else f (acc*n) (n-1)
   in f 1 n;;
```

- La première version n'est pas récursive terminale
- la seconde l'est.
  - dans `f`, le paramètre `acc` est appelé *accumulateur*. Au $i^{\text{ème}}$ appel, *acc* contient le résultat partiellement calculé : $\texttt{acc} = \prod_{k=n}^{n+1-(i-1)} k$.

Programmation récursive efficace

Exemple

Exponentiation « égyptienne » : (récursion non terminale)

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{\frac{n}{2}})^2 & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * (a^{\frac{n-1}{2}})^2 & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let puiss a n =
    let rec puiss' = function
        0                    -> 1
      | n when n mod 2=0 -> puissPaire n
      | n                    -> puissImpaire n
    and puissPaire n =
         (let x = (puiss' (n/2)) in x*x)
    and puissImpaire n =
         (let x = (puiss' (n/2)) in a*x*x)
    in if n<0 then failwith "puiss a n: n<0 !"
       else puiss' n;;
```

Programmation récursive efficace

Exemple

Exponentiation « égyptienne » : (récursion terminale)

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^2)^{\frac{n}{2}} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ a * (a^2)^{\frac{n-1}{2}} & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

```
# let puiss a n =
    let rec puiss' acc a = function
        0                    -> acc
      | n when n mod 2=0 -> puissPaire acc a n
      | n                    -> puissImpaire acc a n
    and puissPaire acc a n   =
         puiss' acc (a*a) (n/2)
    and puissImpaire acc a n =
         puiss' (a*acc) (a*a) ((n-1)/2)
    in if n<0 then failwith "puiss a n: n<0 !"
       else puiss' 1 a n;;
```

Programmation récursive efficace

Une fonction récursive terminale se transforme facilement en boucle :

```
fonction f(n)
  si (condition)
  alors
     instructions arrêt
  sinon
     instructions
     f(g(n))
```

```
fonction f'(n)
  tant que (non condition)
    instructions
    n <- g(n)
  instructions arrêt
```

Exemple

```
unsigned long fact(unsigned n) {
  unsigned long int res=1UL;
  while (n>0) {
    res *= n;
    --n;
  }
  return res;
}
```

Programmation récursive efficace

OCaml transforme de manière automatique les fonctions récursives terminales en itérations sur la plateforme d'exécution.

Il faut donc,

- dans la mesure du possible (ça ne l'est pas toujours),
- dans la mesure du raisonnable
  - la performance (taille de la pile d'exécution, temps d'exécution) peut ne pas être un critère important
  - la lisibilité peut en souffrir
  - ça peut être difficile

écrire des fonctions récursives terminales.

Il faut également éviter d'empiler des arguments inutiles.

Avantages de la récursivité

Elle permet :

- des définitions souvent naturelles, courtes et lisibles ;
- des définitions sur lesquelles des propriétés peuvent naturellement être montrées par récurrence ;
- OCaml sait optimiser l'évaluation des définitions récursives terminales ;
- tout ce qui s'écrit en impératif s'écrit en récursif (et réciproquement) ;
- facilite le « diviser pour régner » (exemples : exponentiation égyptienne, tri fusion, etc.)
- facilite le *back-tracking* (exemples : 8 dames, cavalier, etc.)

Tri fusion

Trier une suite s d'éléments $\rightarrow s'$

Si $|s| < 2$ il n'y a rien à faire.
Sinon :

- couper s en deux parties s_1 et s_2 de tailles identiques ± 1
- Trier (induction sur la taille de la suite) $s_1 \rightarrow s'_1$
- Trier (induction sur la taille de la suite) $s_2 \rightarrow s'_2$
- Construire s' par fusion de s'_1 et s'_2

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' :
- s'_1 : $3, 27, 38, 43$
- s'_2 : $9, 10, 82$

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : 3
- s'_1 : $27, 38, 43$
- s'_2 : $9, 10, 82$

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9$
- s'_1 : $27, 38, 43$
- s'_2 : $10, 82$

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9, 10$
- s'_1 : $27, 38, 43$
- s'_2 : 82

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9, 10, 27$
- s'_1 : $38, 43$
- s'_2 : 82

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9, 10, 27, 38$
- s'_1 : 43
- s'_2 : 82

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9, 10, 27, 38, 43$
- s'_1 :
- s'_2 : 82

Tri fusion

Fusion de s'_1 et $s'_2 \rightarrow s'$:

Si s'_1 ou s'_2 est vide, s' est celle qui n'est pas vide.
Sinon :

- $\min s' = \min(\min s'_1, \min s'_2)$
- le reste de s' s'obtient par fusion du reste de s'_1 et du reste de s'_2 (induction sur $|\text{reste de } s'_1| + |\text{reste de } s'_2|$)

Exemple :

- s' : $3, 9, 10, 27, 38, 43, 82$
- s'_1 :
- s'_2 :

Tri fusion

Fusion de deux listes triées :

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

Tri fusion

Version récursive terminale de la fusion :

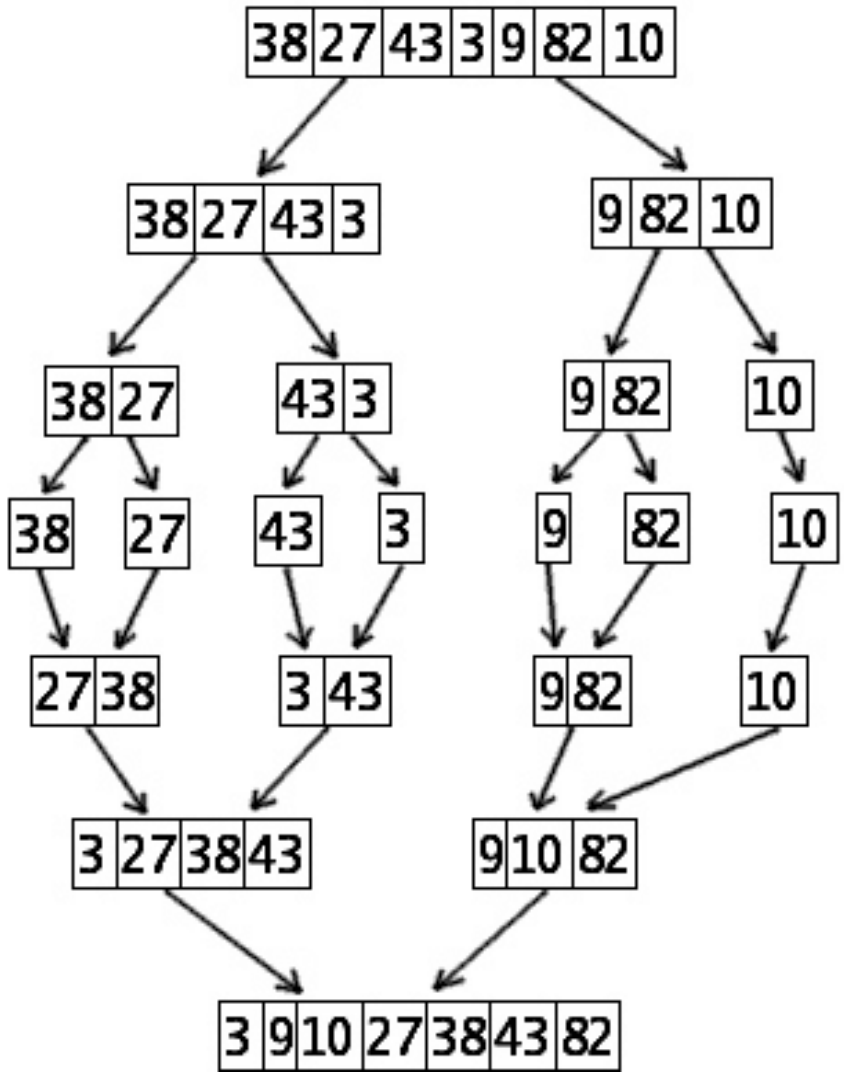
```
# let fusion l l' =
    let rec fusion_rec acc l l' = match (l,l') with
      [],[] -> List.rev acc
    | [],x  -> (List.rev acc)@x
    | x,[]  -> (List.rev acc)@x
    | x::x',y::y' ->
            if x<y then fusion_rec (x::acc) x' l'
            else fusion_rec (y::acc) l y'
    in fusion_rec [] l l'
;;
val fusion : 'a list -> 'a list -> 'a list = <fun>
```

Tri fusion

```
# let partage l =
    let rec partage_rec (l1,l2) = function
        []    -> (l1,l2)
      | x::l -> partage_rec (l2,x::l1) l
    in partage_rec ([],[]) l
;;
val partage : 'a list -> 'a list * 'a list = <fun>

# let rec tri_fusion l =
    if List.length l < 2 then l
    else let (l1,l2)=partage l
         in fusion (tri_fusion l1) (tri_fusion l2)
;;
val tri_fusion : 'a list -> 'a list = <fun>
```

Tri fusion



source : wikipedia

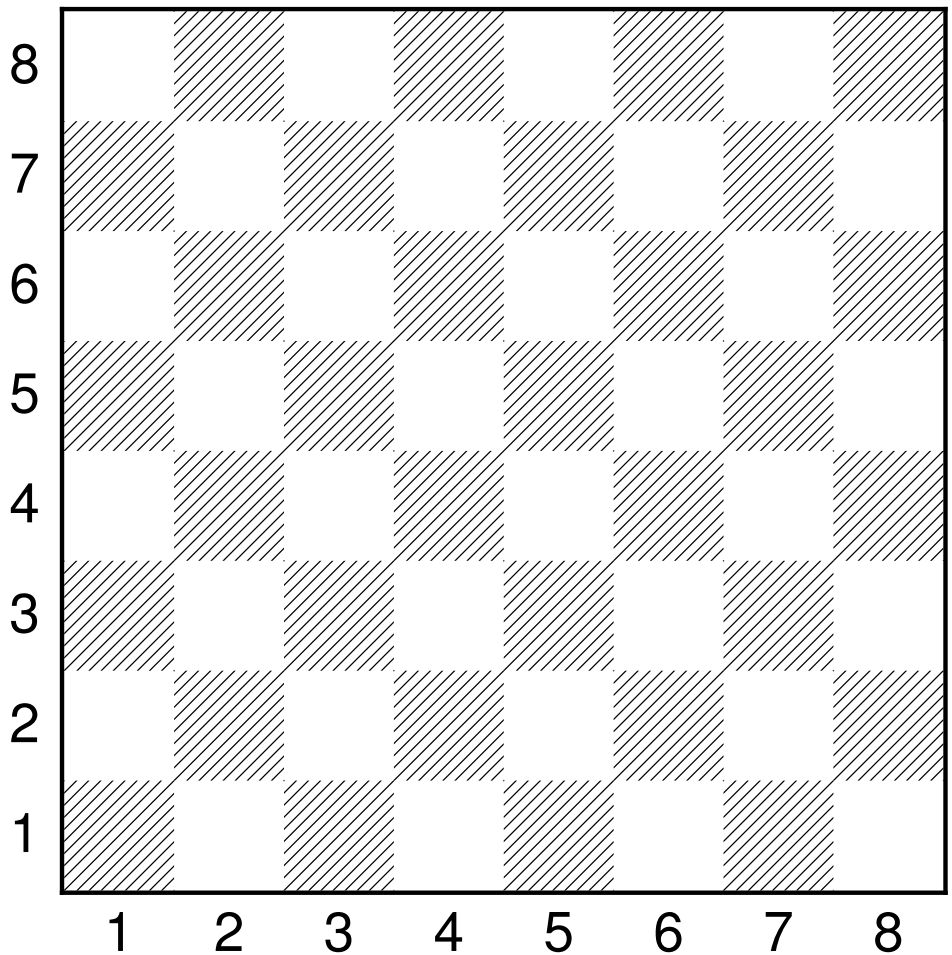
Tri fusion

En prenant la relation d'ordre en paramètre :

```
# let tri_fusion inf l =
    let rec fusion_rec acc l l' = match (l,l') with
      [],[] -> List.rev acc
    | [],x  -> (List.rev acc)@x
    | x,[]  -> (List.rev acc)@x
    | x::x',y::y' ->
       if inf x y then fusion_rec (x::acc) x' l'
       else fusion_rec (y::acc) l y'
    in let rec tri_fusion_rec l =
       if List.length l < 2 then l
       else let (l1,l2)=partage l
    in fusion_rec [] (tri_fusion_rec l1)
                                  (tri_fusion_rec l2)
  in tri_fusion_rec l
;;
val tri_fusion : ('a -> 'a -> bool) -> 'a list -> 'a list = <f
# tri_fusion (fun x y -> x<y) [38;27;43;3;9;82;10] ;;
- : int list = [3; 9; 10; 27; 38; 43; 82]
# tri_fusion (fun x y -> x>y) [38;27;43;3;9;82;10] ;;
- : int list = [82; 43; 38; 27; 10; 9; 3]
```

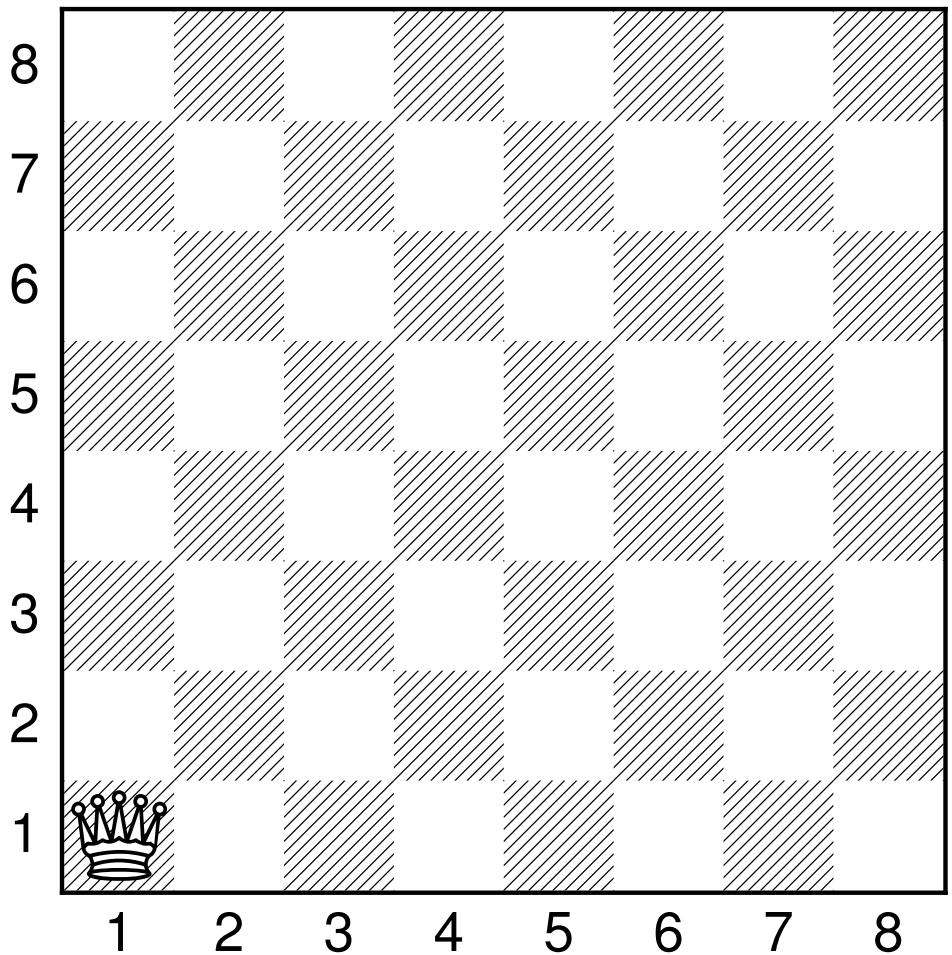
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



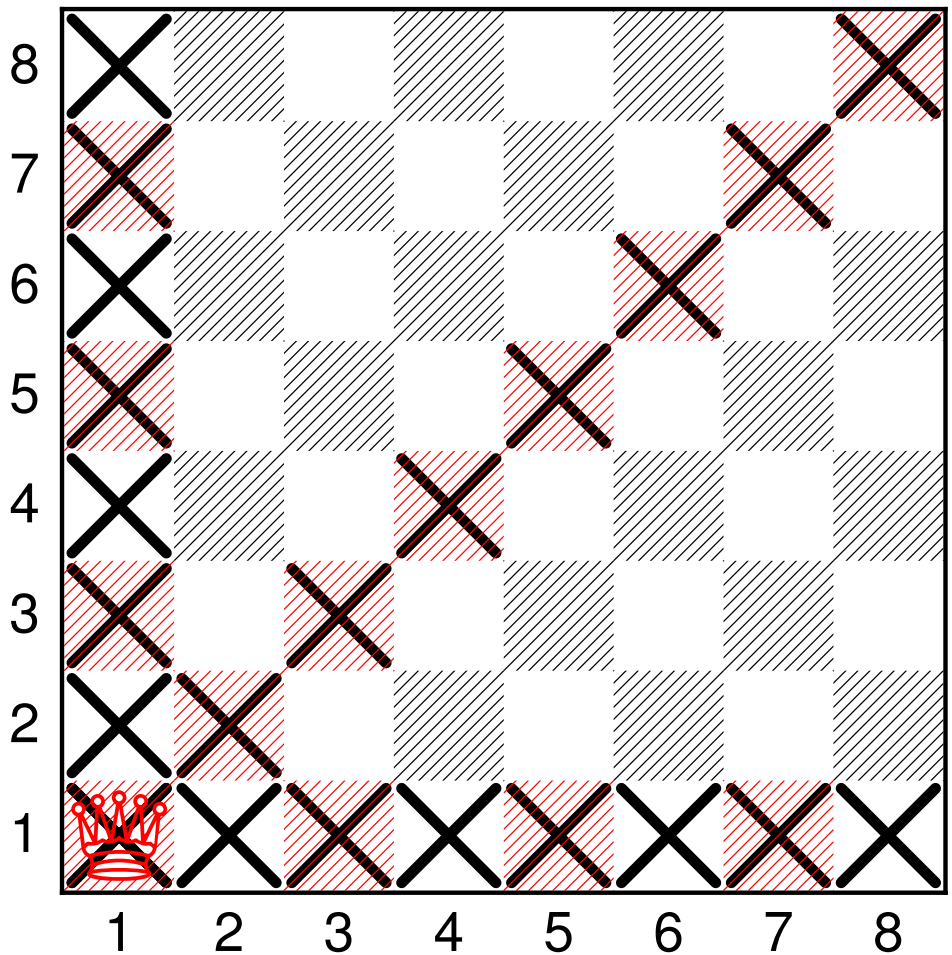
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



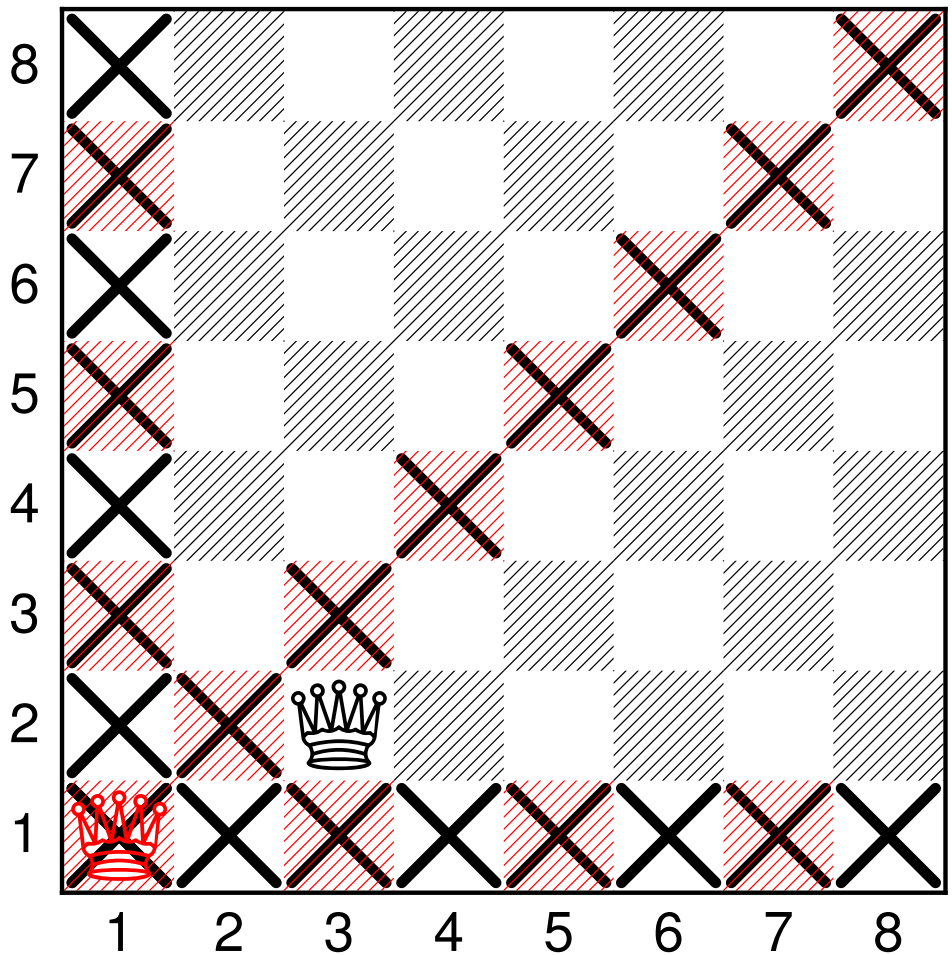
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



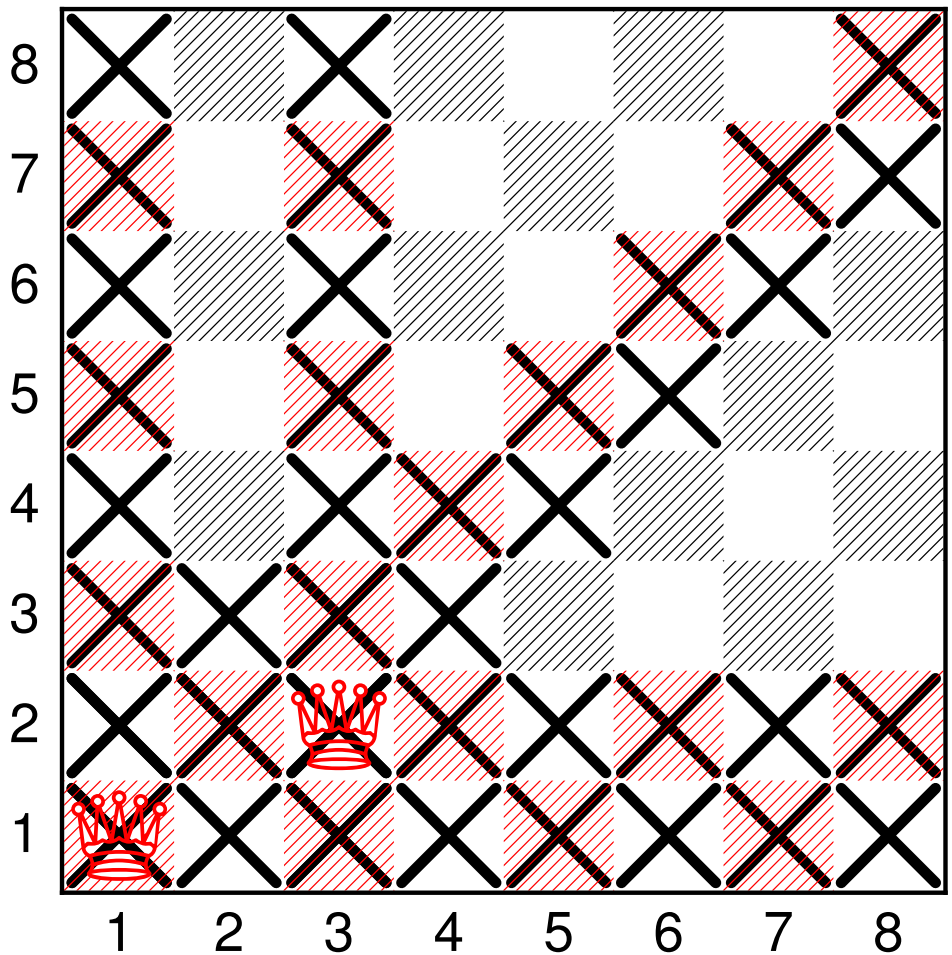
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



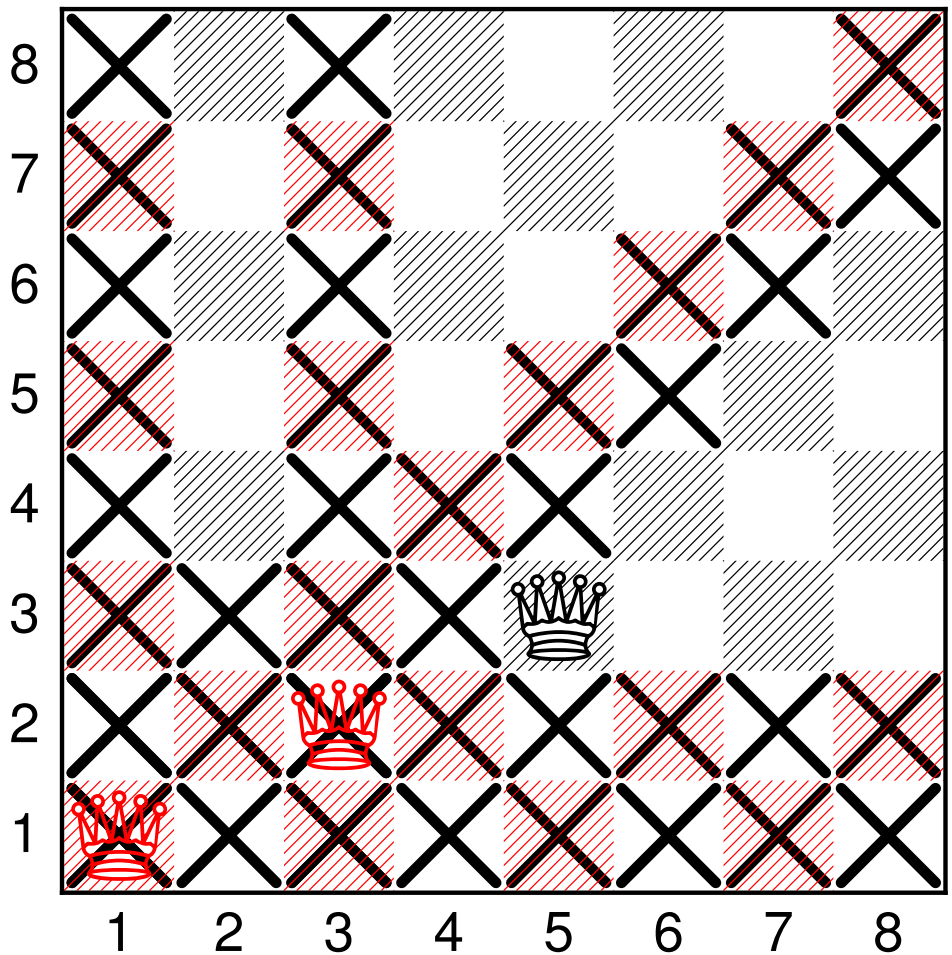
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



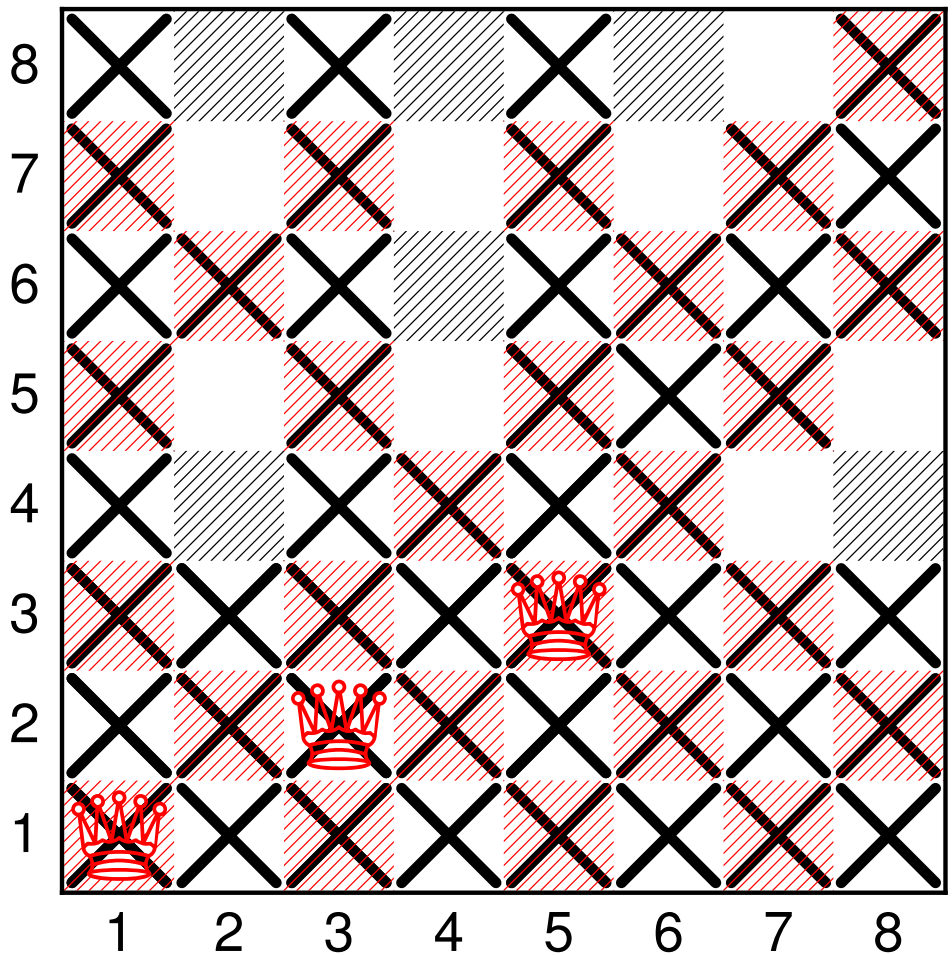
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



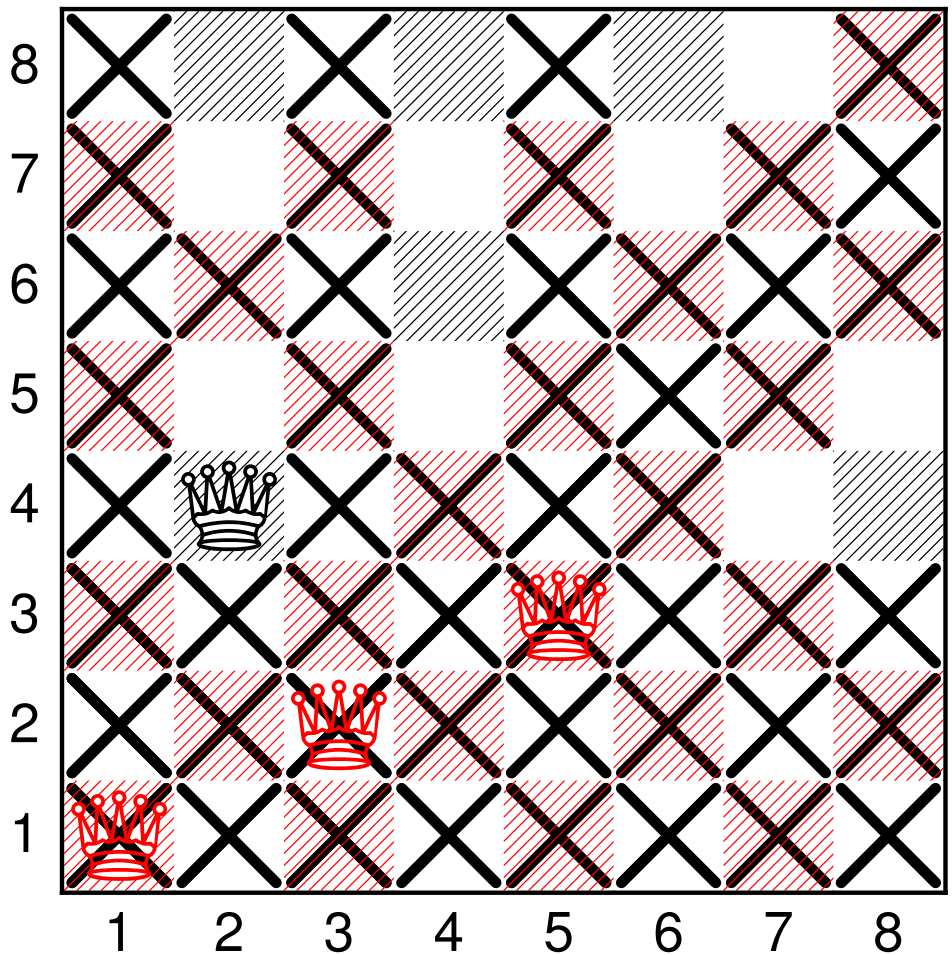
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



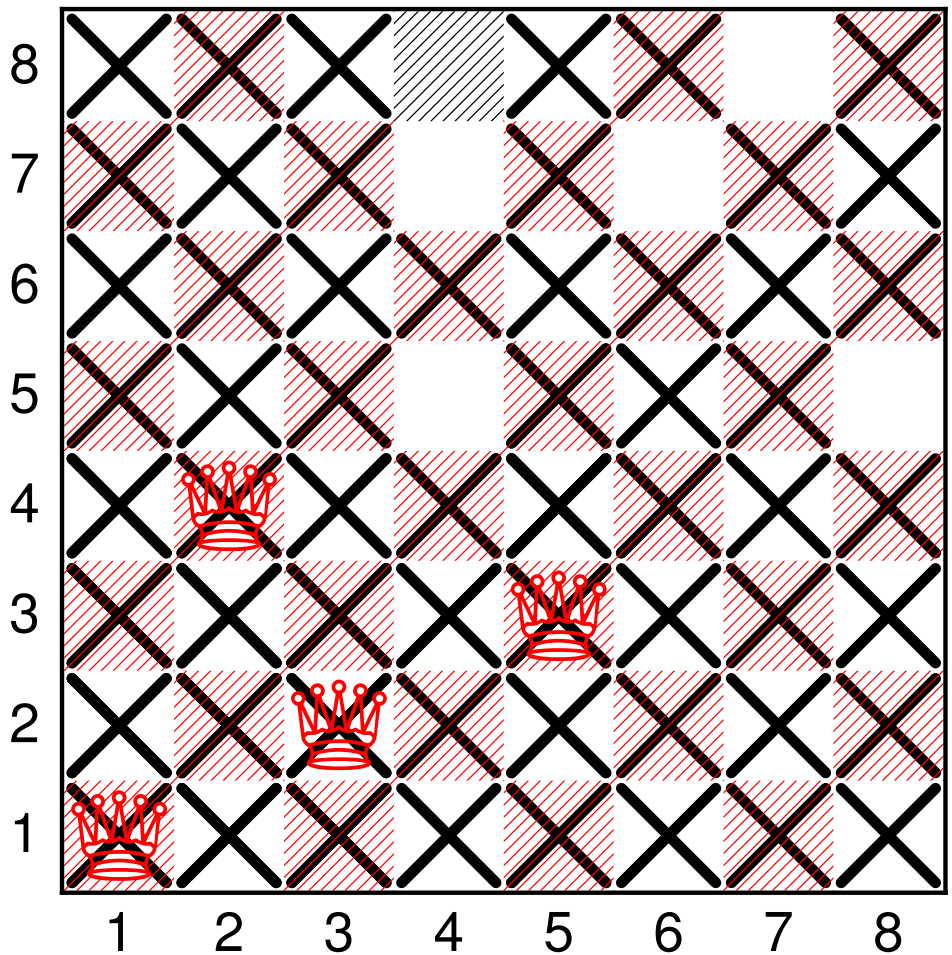
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



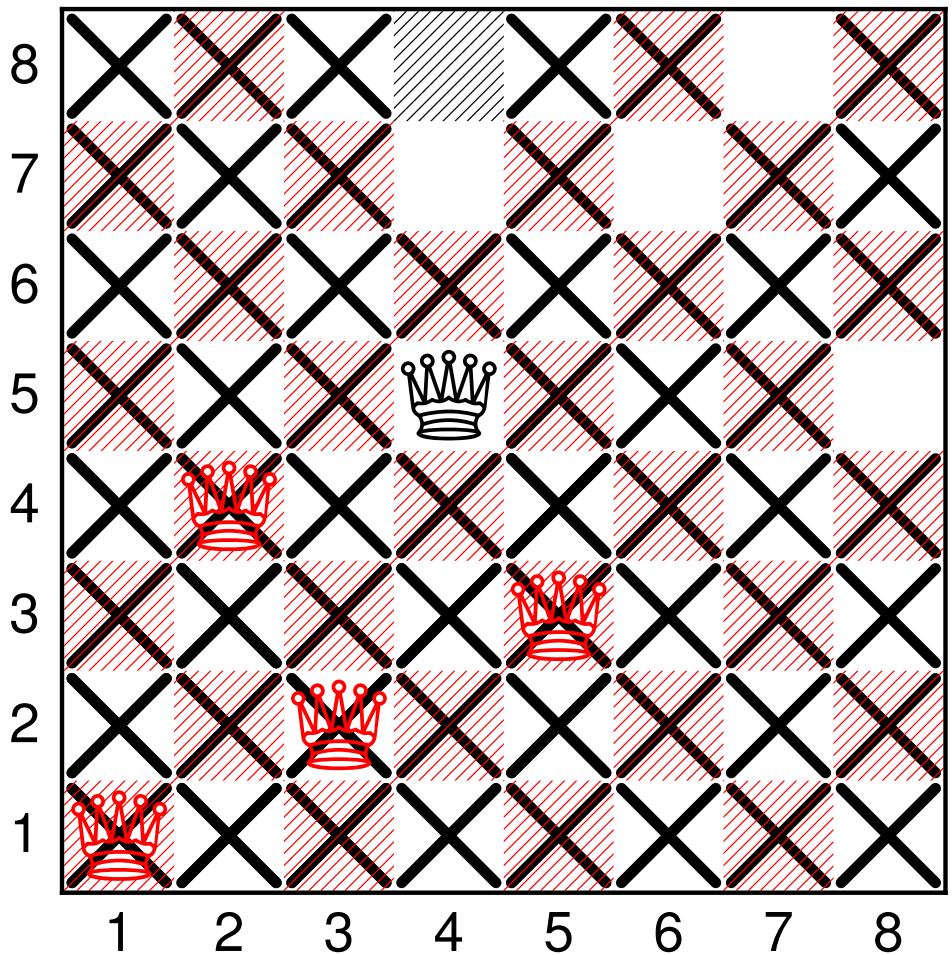
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



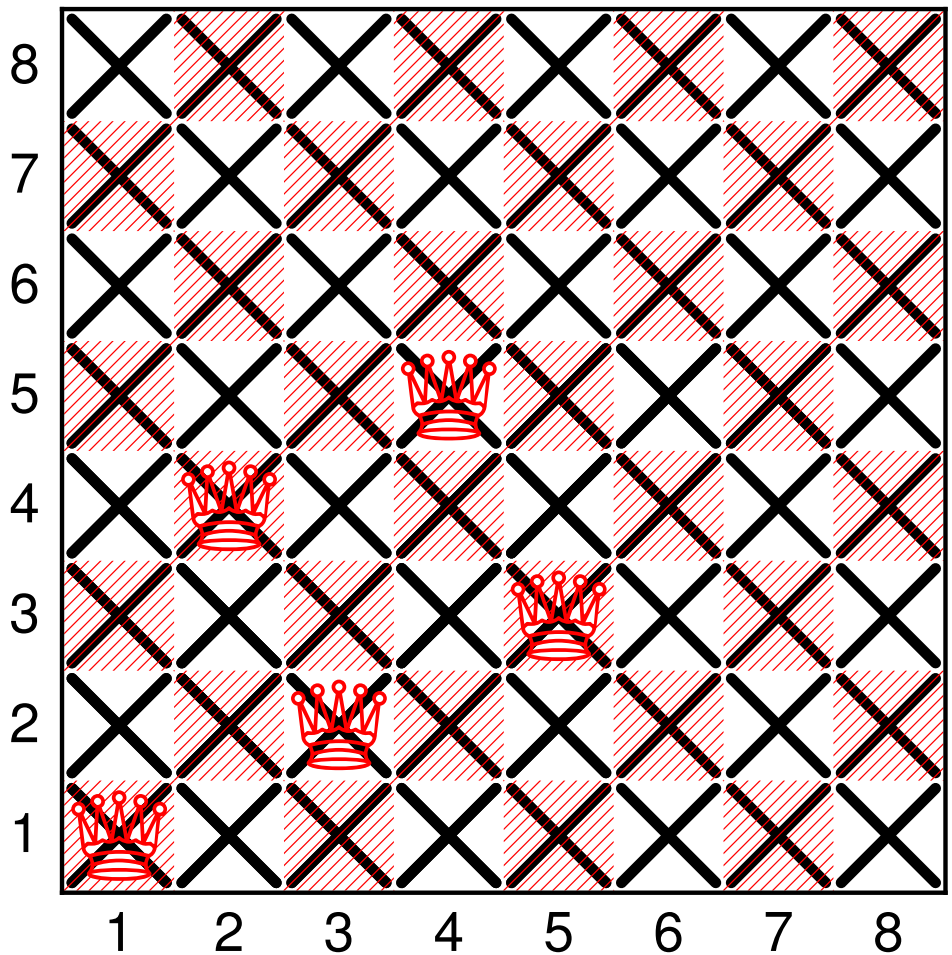
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



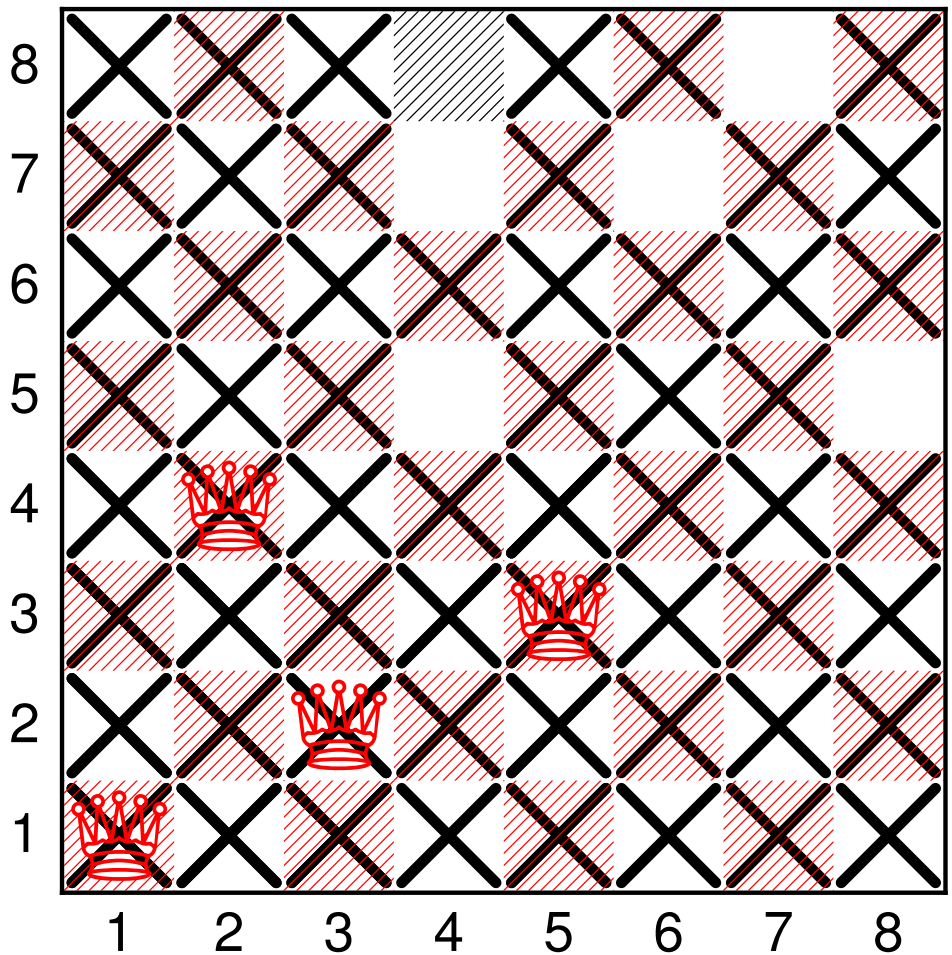
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



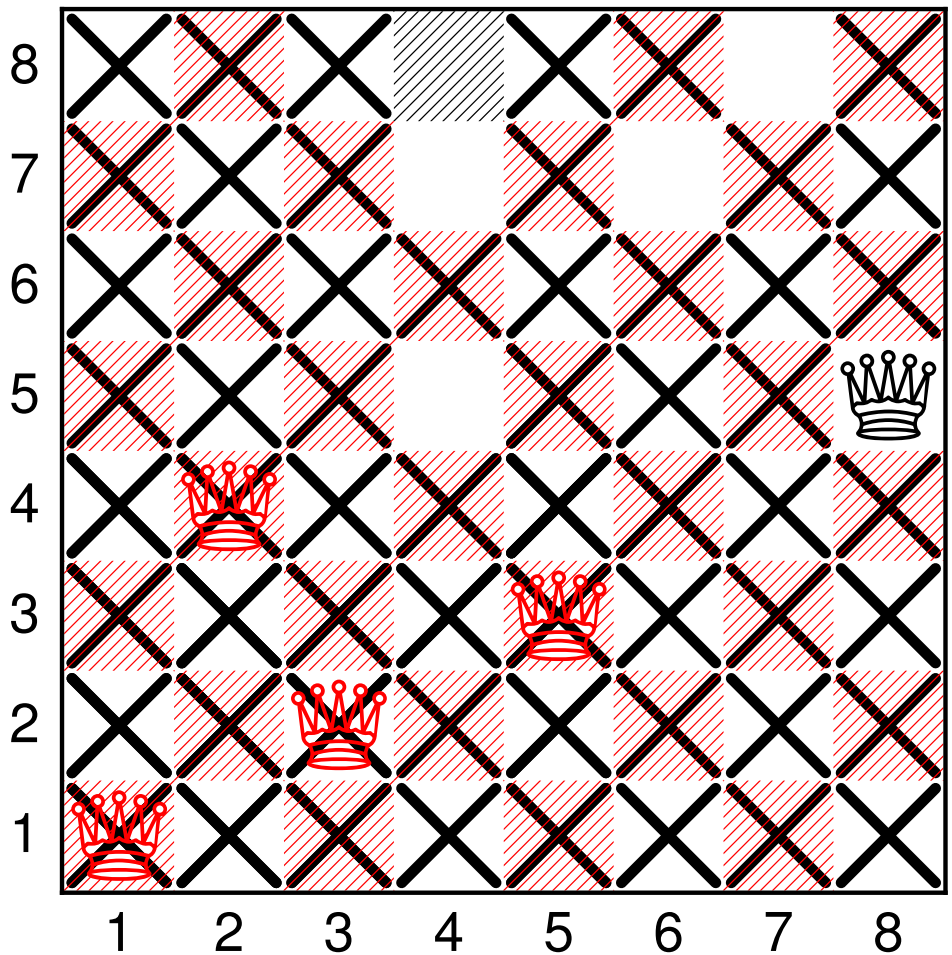
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



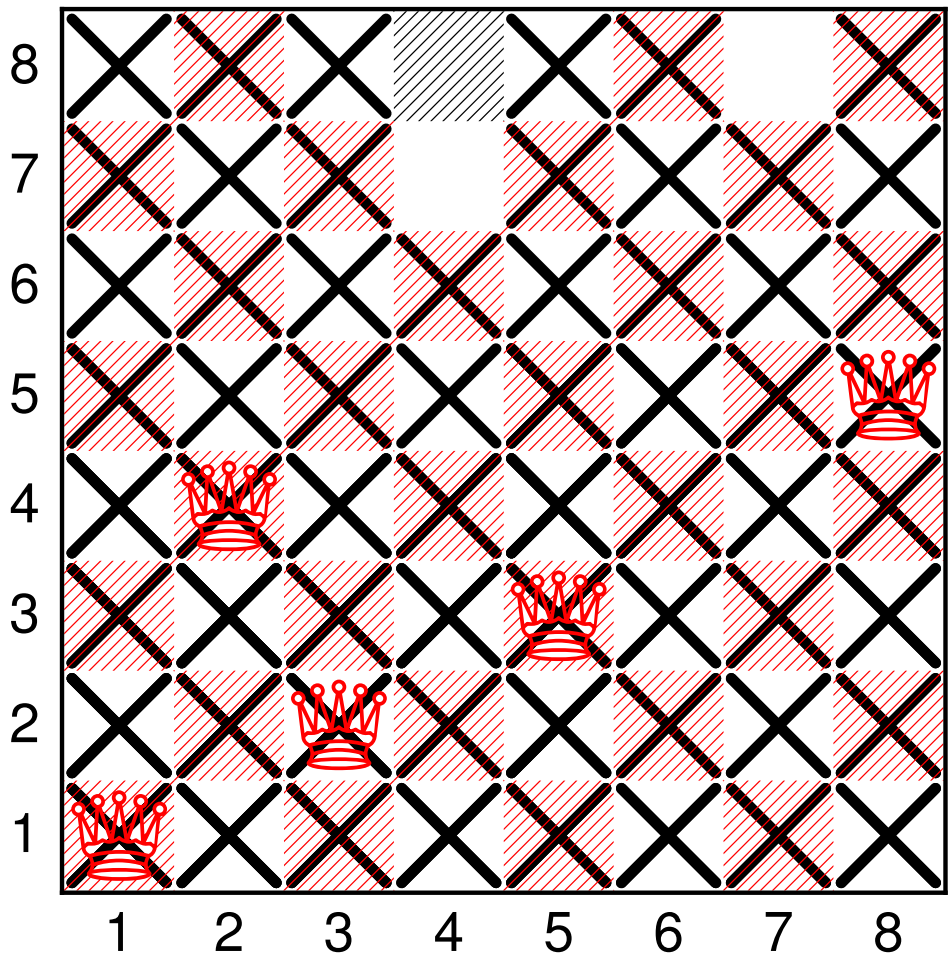
Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres



Le problème des 8 dames

Placer 8 dames sur un échiquier, chacune hors de portée des autres

etc

Le problème des 8 dames

Principe :

- Ordonner les cases par une relation successeur $s(x, y)$
- Placer une dame sur la première case
- Tant qu'il y a des dames à placer
  S'il existe une case libre après le dernier choix
    Y placer une dame
  Sinon
    Le dernier placement (x, y) était mauvais : *back-tracking*
    En retirer la dame
    Si $s(x, y)$ est défini
      Y placer la dame
    Sinon
      S'il y a encore des dames à retirer
        Continuer le *back-tracking*
      Sinon
        Il n'y a pas de solution

Il s'écrit très naturellement de manière récursive.

Le problème des 8 dames

Organisation du damier :

- système de coordonnées totalement ordonné
- une fonction `succ` calculant la coordonnée suivante (si elle existe)

```
# let succ = function
    (8,8) -> (0,0)
  | (8,y) -> (1,y+1)
  | (x,y) when x>=1 && x<=8 && y>=1 && y<=8 -> (x+1,y)
  | _ -> failwith "succ: coordonnées incorrectes"
;;
val succ : int * int -> int * int = <fun>
# succ (8,8);;
- : int * int = (0, 0)
# succ (8,3);;
- : int * int = (1, 4)
# succ (4,3);;
- : int * int = (5, 3)
```

Le problème des 8 dames

- Vérifie que les coordonnées (x, y) sont compatibles avec celles qui sont dans *l*
- Remarque : cette fonction est récursive terminale grâce à l'évaluation paresseuse

```
# let compat l (x,y) =
    let rec compat_rec = function
      | [] -> true
      | (x',y')::l -> x!=x' && y!=y'
                      && (abs (x-x') != abs (y-y'))
                      && compat_rec l
    in compat_rec l
;;
val compat : (int * int) list -> int * int -> bool = <fu
```

Le problème des 8 dames

Trouver la prochaine coordonnée $(x', y') \geq (x, y)$ compatible avec les dames déjà positionnées (dans l) = appliquer `succ` à partir de (x, y) jusqu'à trouver $(x', y') \geq (x, y)$ compatible avec l

Généralisation :

- Entrée :
  - une valeur x ;
  - une condition c ;
  - une fonction f
- Sortie : $f^n(x)$ vérifiant c avec n plus petit possible.

```
let rec f_until_c f c x =
  let rec f_until_c' x =
    if c x then x
    else f_until_c' (f x)
  in f_until_c' x
;;
val f_until_c : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>
f_until_c ( (+) 1) (function x -> x>= 10 && (x mod 3==0) ) 1;;
- : int = 12
```

Le problème des 8 dames

Trouver la prochaine coordonnée $(x', y') \geq (x, y)$ compatible avec les dames déjà positionnées (dans l) = appliquer `succ` à partir de (x, y) jusqu'à trouver $(x', y') \geq (x, y)$ compatible avec l

Généralisation :

- Entrée :
  - une valeur x ;
  - une condition c ;
  - une fonction f
- Sortie : $f^n(x)$ vérifiant c avec n plus petit possible.

```
# let next_compatible_position (x,y) l =
  try f_until_c succ (compat l) (x,y)
  with Failure _ -> (0,0)
;;
val next_compatible_position : int * int -> (int * int) list -> int *
# next_compatible_position (succ (3,2)) [(3,2);(1,1)];;
- : int * int = (5, 3)
# let next_compatible_position' = function
    [] -> (1,1)
  | (x::_) as l -> next_compatible_position x l
;;
```

Le problème des 8 dames

Trouver la prochaine coordonnée $(x', y') \geq (x, y)$ compatible avec les dames déjà positionnées (dans l) = appliquer `succ` à partir de (x, y) jusqu'à trouver $(x', y') \geq (x, y)$ compatible avec l

Généralisation :

- Entrée :
  - une valeur x ;
  - une condition c ;
  - une fonction f
- Sortie : $f^n(x)$ vérifiant c avec n plus petit possible.

```
# next_compatible_position' [];;
- : int * int = (1, 1)
# next_compatible_position' [(1,1)];;
- : int * int = (3, 2)
# next_compatible_position' [(3,2);(1,1)];;
- : int * int = (5, 3)
# next_compatible_position' [(5,3);(3,2);(1,1)];;
- : int * int = (2, 4)
# next_compatible_position' [(2,4);(5,3);(3,2);(1,1)];;
- : int * int = (4, 5)
# next_compatible_position' [(4,5);(2,4);(5,3);(3,2);(1,1)];;
- : int * int = (0, 0)
```

Le problème des 8 dames

Pour finir, deux fonctions mutuellement récursives :

- `queens` *l* :
  - si elle existe, retourne une solution au problème des 8 dames avec des dames déjà positionnées dans *l* ;
  - retourne `[]` sinon.
- `backward_queens` *l* :
  - (*back-tracking*) revient sur le dernier choix qui a été fait (dans *l*), en annulant plusieurs choix si nécessaire

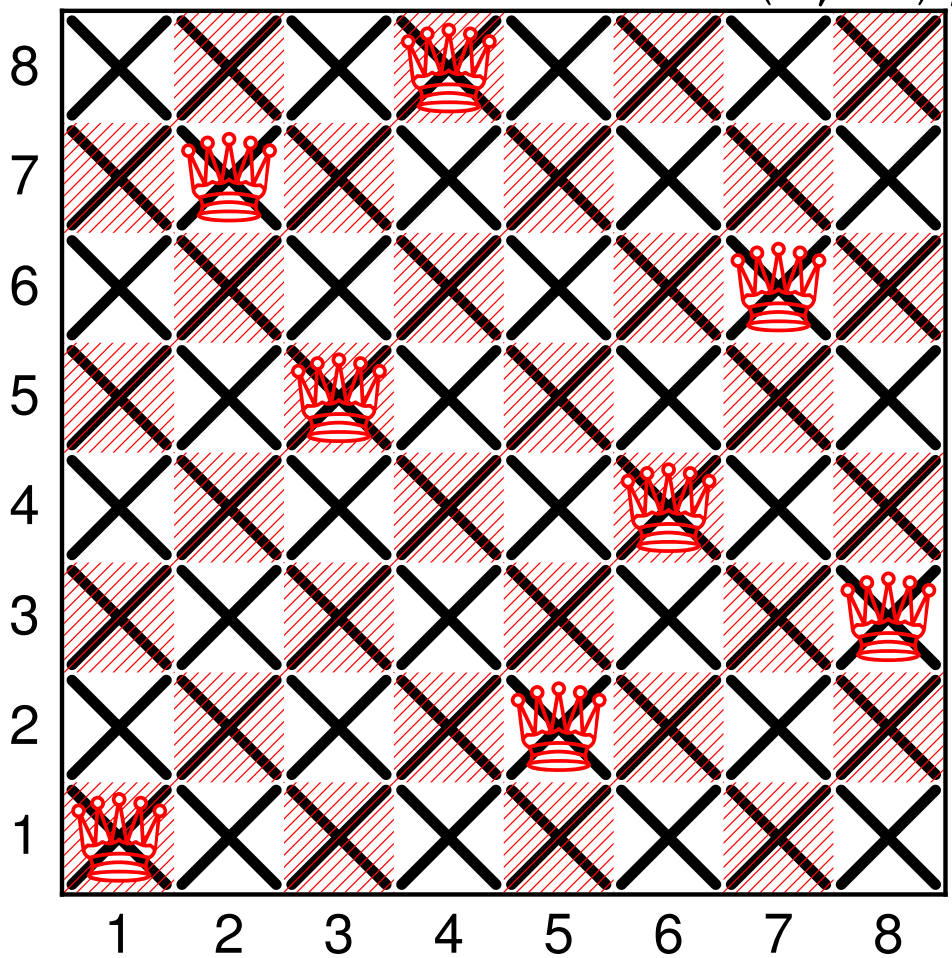
Le problème des 8 dames

```
# let rec queens = function
  | [] -> queens [(1,1)]
  | l  -> if List.length l = 8 then l
          else let c = next_compatible_position' l
               in if c = (0,0) then backward_queens l
                  else queens (c::l)
and backward_queens = function
  | []    -> []
  | c::l' -> let n = succ c
             in if n = (0,0)
                then backward_queens l'
                else let n' = next_compatible_position n l'
                     in if n' = (0,0) then backward_queens l'
                        else queens (n'::l')
;;
val queens : (int * int) list -> (int * int) list = <fun>
val backward_queens : (int * int) list -> (int * int) list = <fun>

# queens [];;
- : (int * int) list =
[(4, 8); (2, 7); (7, 6); (3, 5); (6, 4); (8, 3); (5, 2); (1, 1)]
```

Le problème des 8 dames

```
# queens [];;
- : (int * int) list = [(4, 8); (2, 7); (7, 6); (3, 5);
                        (6, 4); (8, 3); (5, 2); (1, 1)]
```



Le problème des 8 dames : bien fondé de la récurrence

À chaque appel récursif de `queens`, le nombre de dames bien placées n'augmente pas nécessairement (il peut même diminuer). Pour montrer la terminaison :

- numérotons 1 la plus petite case, 2 la suivante, etc
  numéros de cases : de 1 à 64
- considérons le nombre

$$p = c_1 65^7 + c_2 65^6 + \cdots + c_8 65^0$$

  où c_i est le numéro de la case sur laquelle la i^e dame a été placée ($c_i = 0$ si la i^e dame n'a pas encore été placée)
- au premier appel de `queens`, p vaut 0
- à chaque appel de `queens`, p augmente strictement
- p est majoré par $64 * 65^7$ (1^e dame placée en haut à droite)
- et donc, la récurrence termine.

Un exemple : les arbres binaires de recherche

Un arbre binaire de recherche (ABR) est soit

- l'arbre vide
- un nœud contenant une valeur v, un fils gauche g et un fils droit d tels que
  - g et v soient des ABR
  - les valeurs de g soient toutes inférieures à v
  - les valeurs de d soient toutes supérieures à v

```
type 'a arbre =
  Vide
| Noeud of 'a*'a arbre*'a arbre
;;
type 'a arbre_recherche = {
  ordre: 'a->'a->bool;
  arbre: 'a arbre
};;
```

Un exemple : les arbres binaires de recherche

```
let hauteur a =
  let rec ht = function
    Vide -> 0
  | Noeud (_,g,d) -> 1+(max (ht g) (ht d))
  in ht a.arbre
;;

let insertion v a =
  let rec ins_rec = function
      Vide -> Noeud (v,Vide,Vide)
    | Noeud (v',g,d) as n ->
       let i = a.ordre v v' and s = a.ordre v' v
       in if i && s then n
  else if i then Noeud (v', (ins_rec g), d)
  else Noeud (v', g, (ins_rec d))
  in { a with arbre = ins_rec a.arbre }
;;
```

Un exemple : les arbres binaires de recherche

```
# let a = { ordre = (<=); arbre = (Vide:int arbre) };;
val a : int arbre_recherche =
   {ordre = <fun>; arbre = Vide}
# (insertion 2 (insertion 9 (insertion 1
      (insertion 4 (insertion 7 (insertion 4
          (insertion 3 (insertion 5 a)))))
  ))).arbre;;
- : int arbre_recherche =
{ordre = <fun>;
 arbre =
  Noeud (5,
   Noeud (3,
     Noeud (1,
       Vide,
       Noeud (2, Vide, Vide)),
     Noeud (4, Vide, Vide)),
   Noeud (7, Vide, Noeud (9, Vide, Vide)))
}
```

Les fonctions d'ordre supérieur

En programmation fonctionnelle, les fonctions sont des valeurs comme les autres.

Elles peuvent donc être des paramètres ou des retours d'autres fonctions.

Une fonction qui en prend une autre en paramètre est dite d'*ordre supérieur*.

L'emploi de fonctions d'ordre supérieur est très courant en programmation fonctionnelle.

- Il permet de mieux découper un programme complexe en fonctions élémentaires, en facilitant la mise en œuvre de l'adage « une fonction, une tâche » ;
- Il améliore la généricité des programmes.

Les fonctions d'ordre supérieur

Exemples

Les fonctions `map` et `rev_map` appliquent une fonction à tous les éléments d'une liste :

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.map (function x->x+1) [1;2;3];;
# - : int list = [2; 3; 4]

# List.rev_map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.rev_map (function x->x+1) [1;2;3];;
- : int list = [4; 3; 2]
```

`map` préserve l'ordre (récursive terminale depuis Ocaml 5.1.0).
`rev_map` renverse l'ordre (récursive terminale).

Les fonctions d'ordre supérieur

Exemples

La fonction `apply` applique une fonction à un argument :

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

La fonction `compose` compose deux fonctions :

```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a)
                         -> 'c -> 'b = <fun>
```

Les fonctions d'ordre supérieur

Faciliter « une fonction, une tâche »
La fonction suivante réalise la somme des éléments d'une liste d'entiers :

```
# let somme l =
    let rec somme_rec acc = function
        [] -> acc
      | x::l -> somme_rec (x+acc) l
    in somme_rec 0 l
;;
```

Elle réalise en fait deux tâches :

- un parcours de la liste, qui ne dépend pas de ce qu'on souhaite calculer ;
- à chaque étape du parcours, on réalise la somme de l'élément « actuel » avec le reste

Une fonction qui calcule le produit des éléments de la liste, ou qui compte le nombre d'éléments de la liste, serait très similaire.

Les fonctions d'ordre supérieur

Faciliter « une fonction, une tâche »
On décompose en écrivant une fonction réalisant le parcours et appliquant une fonction à chaque étape :

```
# let parcours f l base =
    let rec parcours_rec = function
      [] -> base
    | x::l -> f x (parcours_rec l)
    in parcours_rec l
;;
val parcours : ('a -> 'b -> 'b) -> 'a list
                  -> 'b -> 'b = <fun>
```

On a alors

```
let somme l =  parcours ( + ) l 0;;
let produit l = parcours ( * ) l 1;;
let longueur l = parcours (fun _ y -> y+1) l 0;;
```

La fonction `parcours` existe déjà : c'est `List.fold_right`.
Remarque : elle n'est pas récursive terminale.

Les fonctions d'ordre supérieur

Faciliter « une fonction, une tâche »
On peut écrire une fonction de parcours récursive terminale :

```
# let parcours2 f l base =
  let rec parcours_rec acc = function
    [] -> acc
  | x::l -> parcours_rec (f acc x) l
  in parcours_rec base l
;;
val parcours2 : ('a -> 'b -> 'a) -> 'b list
                    -> 'a -> 'a = <fun>
```

On a alors

```
let somme l =  parcours2 ( + ) l 0;;
let produit l = parcours2 ( * ) l 1;;
let longueur l = parcours2 (fun x _ -> x+1) l 0;;
```

La fonction `parcours2` existe déjà (à l'ordre des arguments près) : c'est `List.fold_left`.

Les fonctions d'ordre supérieur

Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` $f\ [e_1; \ldots; e_n]\ a$

- calcule $f\ (e_1\ (f\ e_2\ (\ldots (f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

`List.fold_left` $f\ a\ [e_1; \ldots; e_n]$

- calcule $f\ (\ldots (f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Les fonctions d'ordre supérieur

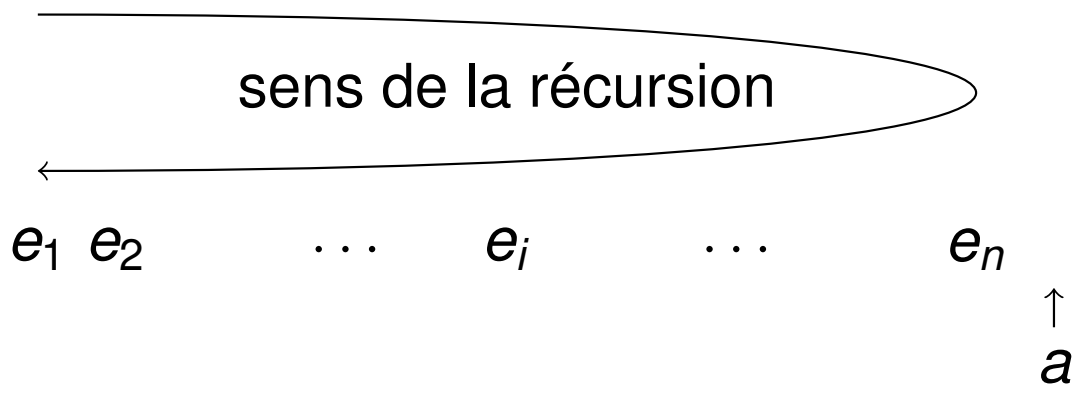
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` $f\ [e_1;\ldots;e_n]\ a$

- calcule $f\ (e_1\ (f\ e_2\ (\ldots(f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

Exemple

```
List.fold_right ( + ) [e1;...;en] a
```



"

Les fonctions d'ordre supérieur

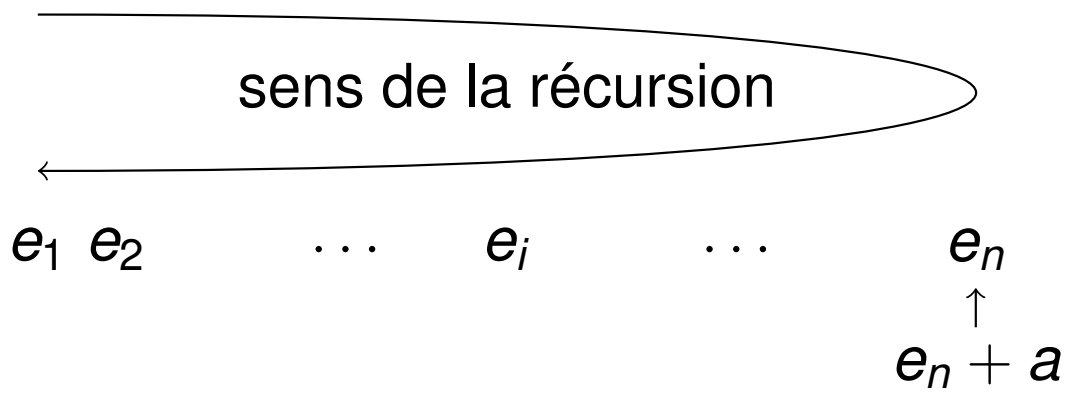
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` $f\ [e_1;\ldots;e_n]\ a$

- calcule $f\ (e_1\ (f\ e_2\ (\ldots(f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

Exemple

`List.fold_right ( + ) [e1;...;en] a`



" :

Les fonctions d'ordre supérieur

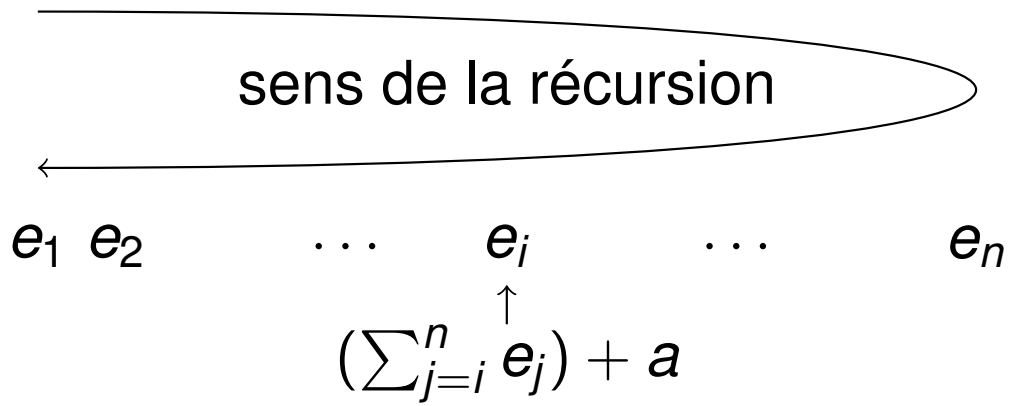
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` $f\ [e_1;\ldots;e_n]\ a$

- calcule $f\ (e_1\ (f\ e_2\ (\ldots(f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

Exemple

```
List.fold_right ( + ) [e1;...;en] a
```



Les fonctions d'ordre supérieur

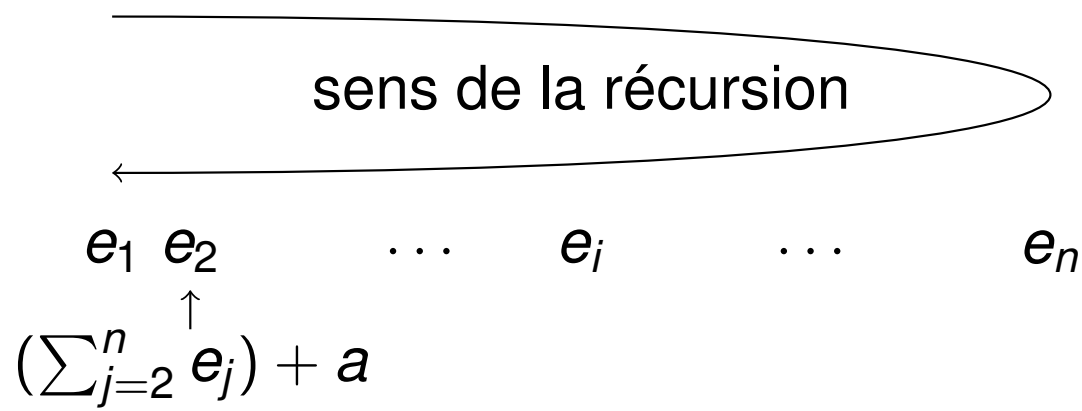
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` $f\ [e_1;\ldots;e_n]\ a$

- calcule $f\ (e_1\ (f\ e_2\ (\ldots(f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

Exemple

```
List.fold_right ( + ) [e1;...;en] a
```



Les fonctions d'ordre supérieur

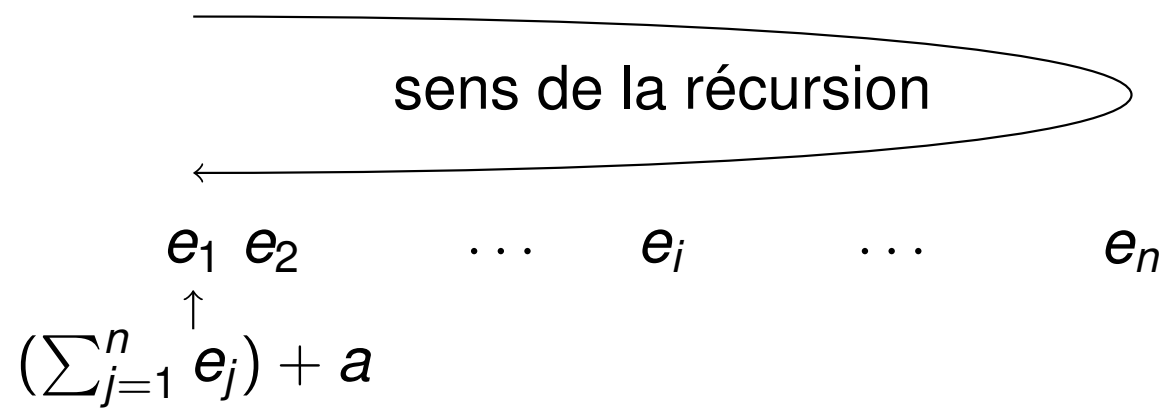
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_right` f $[e_1;\ldots;e_n]$ a

- calcule $f\ (e_1\ (f\ e_2\ (\ldots(f\ e_n\ a)\ldots)))$
- a le type $('a \to' b \to' b) \to' a\ list \to' b \to' b$
- n'est pas récursive terminale

Exemple

`List.fold_right ( + ) [e1;...;en] a`



" : nd

Les fonctions d'ordre supérieur

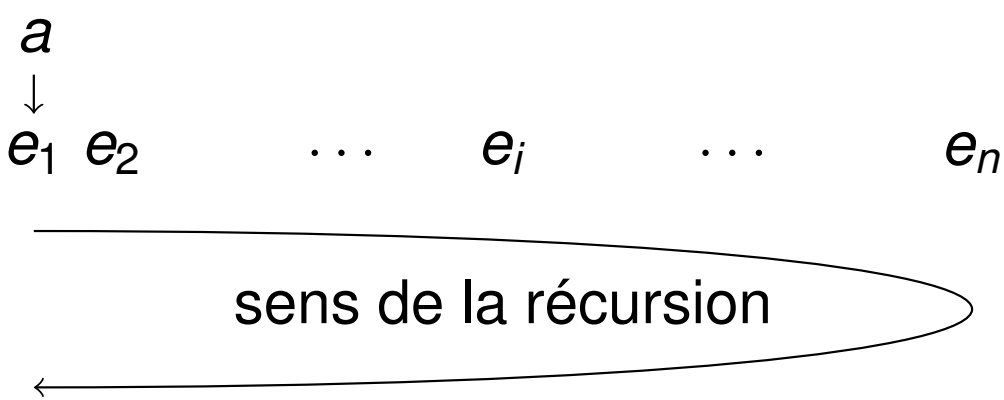
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_left` $f\ a\ [e_1;\ldots;e_n]$

- calcule $f\ (\ldots(f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Exemple

```
List.fold_left ( + ) a [e1;...;en]
```



Les fonctions d'ordre supérieur

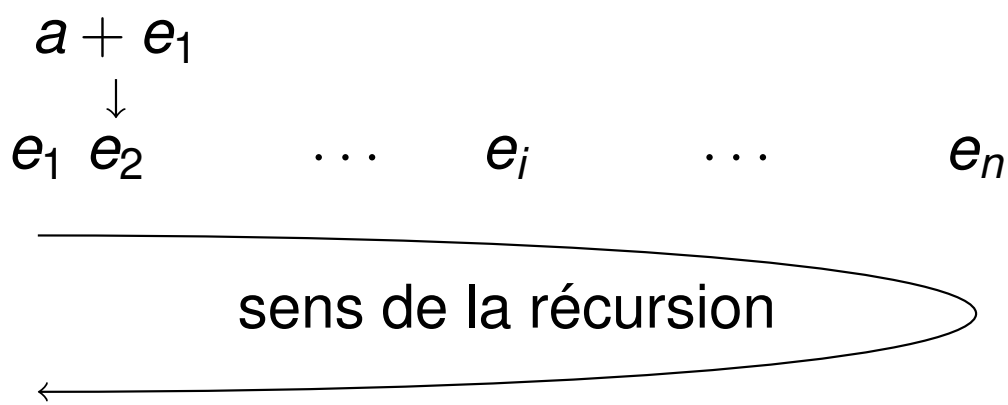
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_left` $f\ a\ [e_1;\ldots;e_n]$

- calcule $f\ (\ldots(f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Exemple

```
List.fold_left ( + ) a [e1;...;en]
```



Les fonctions d'ordre supérieur

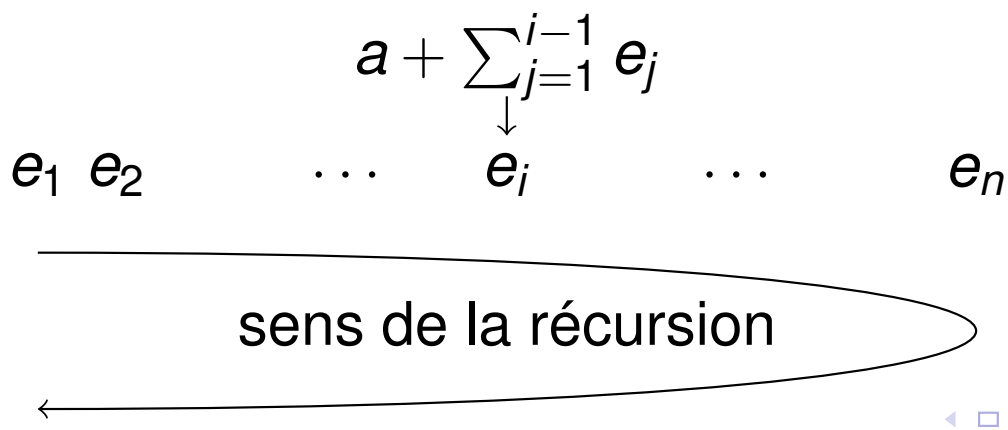
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_left` $f\ a\ [e_1;\ldots;e_n]$

- calcule $f\ (\ldots(f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Exemple

```
List.fold_left ( + ) a [e1;...;en]
```



Les fonctions d'ordre supérieur

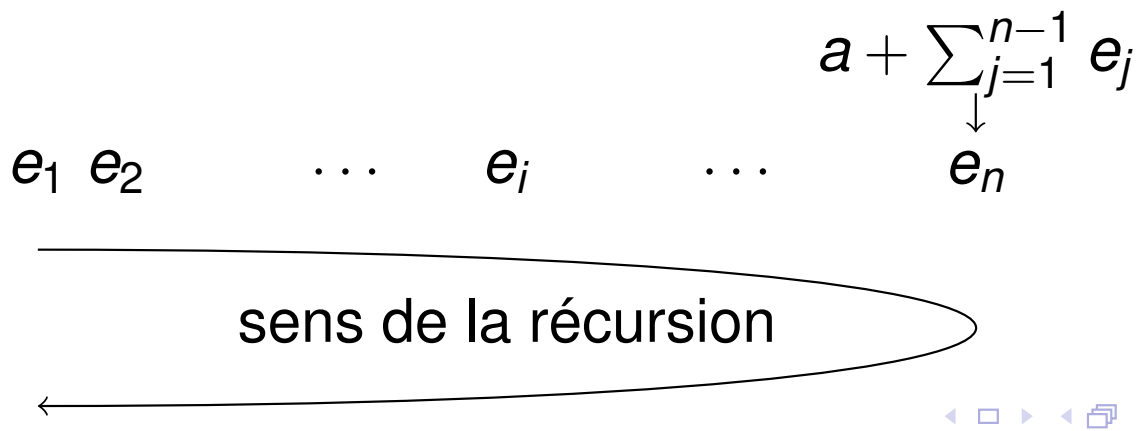
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_left` $f\ a\ [e_1;\ldots;e_n]$

- calcule $f\ (\ldots(f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Exemple

```
List.fold_left ( + ) a [e1;...;en]
```



Les fonctions d'ordre supérieur

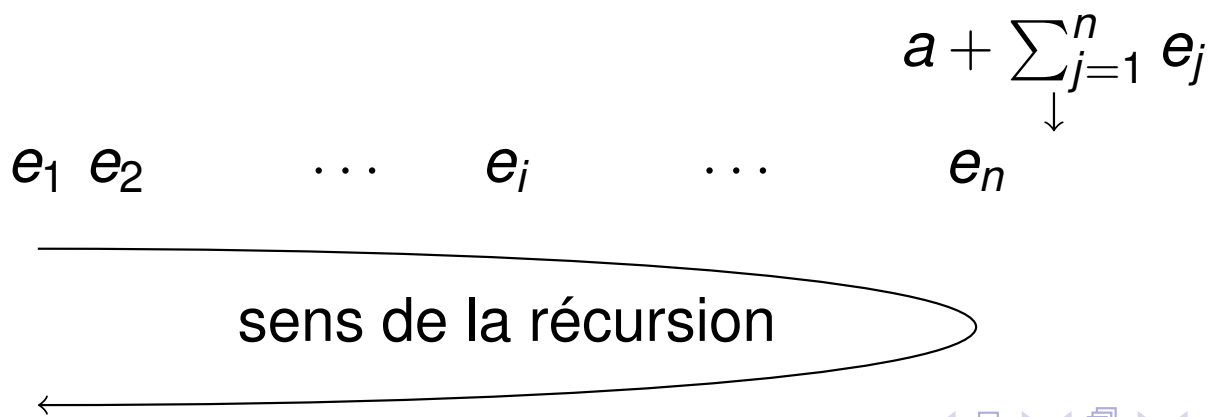
Remarque : il a fallu changer la définition de la fonction de longueur.
En fait, les fonctions `List.fold_right` et `List.fold_left` ne sont équivalentes ni algorithmiquement, ni sémantiquement.

`List.fold_left` $f\ a\ [e_1;\ldots;e_n]$

- calcule $f\ (\ldots(f\ (f\ a\ e_1)\ e_2)\ldots)\ e_n$
- a le type $('a \to' b \to' a) \to' a \to' b\ list \to' a$
- est récursive terminale

Exemple

```
List.fold_left ( + ) a [e1;...;en]
```



Les fonctions d'ordre supérieur

Faciliter « une fonction, une tâche »
Exemple : rechercher un élément dans une liste

```
# let appartient v l =
    parcours2 (fun a x -> a || x=v) l false;;
val appartient : 'a -> 'a list -> bool = <fun>
```

Remarque : la liste est toujours entièrement parcourue.
On peut interrompre le parcours dès que l'élément est trouvé en utilisant une exception :

```
# let appartient v l =
   try parcours2
         (fun _ x -> if x=v
                     then failwith "trouvé!"
                     else false)
         l
         false
   with Failure _ -> true;;
val appartient : 'a -> 'a list -> bool = <fun>
```

Défaut : mise en œuvre du mécanisme des exceptions.
Fonction de recherche déjà existante : `List.mem`

Les fonctions d'ordre supérieur

Exemple : simulation d'une boucle

`boucle` *cond pas v i* :

- itère l'application d'une fonction *i* à une valeur, initialement *v*,
  - en faisant progresser *v* grâce à la fonction *pas*,
  - jusqu'à ce que *cond v* devienne fausse
- retourne toujours *false*
- utilise l'évaluation paresseuse pour la séquentialité (programmation fonctionnelle impure)

```
# let boucle cond pas v i =
  let t x = true
  in let rec boucle_rec v = cond v && (t (i v))
                                   && boucle_rec (pas v)
     in boucle_rec v
;;
val boucle : ('a -> bool) -> ('a -> 'a) -> 'a
                -> ('a -> 'b) -> bool = <fun>
```

Les fonctions d'ordre supérieur

Exemple : simulation d'une boucle

Version sans évaluation paresseuse :

```
# let boucle' cond pas v i =
    let t x = true
    in let rec boucle_rec acc v =
         if cond v then boucle_rec (acc && t (i v)) (pas
         else false
       in boucle_rec true v
;;
val boucle' : ('a -> bool) -> ('a -> 'a) -> 'a
                -> ('a -> 'b) -> bool = <fun>
# boucle' (function i->i<10) ( ( + ) 1 )
          0 print_int;;
0123456789- : bool = false
```

L'opérateur ; de séquentialité

Si e est une expression de type `unit`, e; e' est une expression dont

- la valeur et le type sont ceux de e' ;
- l'évaluation passe par l'évaluation de e d'abord, puis celle de e'.

```
# let x = 4;;
val x : int = 4
# let i=print_int (x+1);print_string "\n"; x+1;;
5
val i : int = 5
```

Comme e est de type `unit`, son évaluation a dans la plupart des cas un effet de bord.

Programmation fonctionnelle impure

Note sur la séquentialité

Dans `let` *nom* `=` *expr* `in` *expr′* `;;` OCaml évalue *expr* avant *expr′*.

Cette propriété d'OCaml peut être utilisée pour simuler la séquentialité

```
# let x = 4;;
val x : int = 4
# let i =
    let () = print_int (x+1)
    in let () = print_string "\n"
       in x+1
  ;;
5
val i : int = 5
```

OCaml dépendant.

`begin` et `end`

Certains parenthésages disgracieux :

```
let f = function
    (0,x) -> (match x with
                 0 -> 1
               | n -> n+1)
  | (1,x) -> ...
```

peuvent être remplacés par l'utilisation de `begin` et `end` :

```
let f = function
    (0,x) -> begin
               match x with
                  0 -> 1
                | n -> n+1
               end
  | (1,x) -> ...
```

Le système de typage

Utilité du typage :

- avertir le programmeur en cas d'incohérence paramètre/argument dans un appel de fonctions ;
- permettre au système de prévoir la place/l'interprétation du retour d'une fonction (sûreté d'exécution).

Le système de typage :

- sait déterminer le type des constantes ;
- mémorise le type de la valeur associée à chaque nom symbolique et opérateur ;
- utilise ces informations pour calculer un type pour chaque expression ;
- le typage est réalisé avant évaluation des expressions pour garantir qu'il n'y aura pas de problème de type pendant l'évaluation ;
- le type calculé est le plus général possible ;
- il peut contenir des inconnues de type (α, β, ...) notées `'a`, `'b`, ...(*polymorphisme*).

Le système de typage

```
# let id = function x -> x;;
val id : 'a -> 'a = <fun>
# id 5;;
- : int = 5
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst (4,'a');;
- : int = 4
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply fst (4,'a');;
- : int = 4
```

Typage monomorphe explicite

Sans inconnue de type : les ambiguïtés de typage sont explicitement levées par le programmeur.
Possible en OCaml (mais nous éviterons)

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 5;;
- : int = 5
# id 'a';;
- : char = 'a'
# let id' (x:int) = x;;
val id' : int -> int = <fun>
# id' 5;;
- : int = 5
# id' 'a';;
Error: This expression has type char but an expression
         was expected of type int
```

Le système de typage : typage monomorphe explicite

Pas d'inconnue de type, type des paramètres déclarés.
Simplification : types de base $\{\texttt{bool}, \texttt{int}\}$
Γ : environnement de types qui

- à des noms associe des types ;
- à chaque opérateur binaire $\oplus$ associe son type.

$$(\mathit{CBool}) : \vdash \text{constante booléenne} : \texttt{bool}$$

$$(\mathit{CInt}) : \vdash \text{constante entière} : \texttt{int}$$

$$(\mathit{Var}) : \Gamma \vdash a : \Gamma(a)$$

$$(\mathit{Op}) : \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau' \qquad \Gamma \vdash \oplus : \tau \to \tau' \to \tau''}{\Gamma \vdash e \oplus e' : \tau''}$$

$$(\mathit{App}) : \frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

$$(\mathit{Fun}) : \frac{\Gamma \cup \{a : \tau\} \vdash e : \tau'}{\Gamma \vdash \texttt{function}\ (a : \tau) \to e : \tau \to \tau'}$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int}} \quad \dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}$$

$$\vdash \texttt{function}\ (f : \texttt{int} \to \texttt{int}) \to (\texttt{function}\ (x : \texttt{int}) \to f\ x) : (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$$

Calcul de type :

$$\vdash \texttt{function}\ (f : \texttt{int} \to \texttt{int}) \to (\texttt{function}\ (x : \texttt{int}) \to f\ x) : \quad$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{}{f\texttt{:int}\to\texttt{int},x\texttt{:int}\vdash f\texttt{:int}\to\texttt{int}} \quad \dfrac{}{f\texttt{:int}\to\texttt{int},x\texttt{:int}\vdash x\texttt{:int}}}{f\texttt{:int}\to\texttt{int},x\texttt{:int}\vdash f\ x\texttt{:int}}}{f\texttt{:int}\to\texttt{int}\vdash\texttt{function}\ (x\texttt{:int})\to f\ x\texttt{:int}\to\texttt{int}}$$

$$\vdash \texttt{function}\ (f : \texttt{int} \to \texttt{int}) \to (\texttt{function}\ (x : \texttt{int}) \to f\ x) : (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$$

Calcul de type :

$$\dfrac{\dfrac{}{f\texttt{:int}\to\texttt{int}\vdash\texttt{function}\ (x\texttt{:int})\to f\ x\texttt{:}}}{\vdash \texttt{function}\ (f : \texttt{int} \to \texttt{int}) \to (\texttt{function}\ (x : \texttt{int}) \to f\ x) :}$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : }$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : }$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) :}$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x:}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) :}$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int} \qquad f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash\texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : }$$

Le système de typage : typage monomorphe explicite

Preuve de type :

$$\dfrac{\dfrac{\dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int}} \quad \dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Calcul de type :

$$\dfrac{\dfrac{\dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f{:}\texttt{int}\to\texttt{int}} \quad \dfrac{}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int},x{:}\texttt{int}\vdash f\ x{:}\texttt{int}}}{f{:}\texttt{int}\to\texttt{int}\vdash \texttt{function}\ (x{:}\texttt{int})\to f\ x{:}\texttt{int}\to\texttt{int}}}{\vdash \texttt{function}\ (f:\texttt{int}\to\texttt{int})\to(\texttt{function}\ (x:\texttt{int})\to f\ x) : (\texttt{int}\to\texttt{int})\to\texttt{int}\to\texttt{int}}$$

Le système de typage : typage monomorphe explicite

Théorème

Si $\vdash e : t$ *et si* e *s'évalue en* v*, alors* v *a type* t*.*

De plus, s'il existe v *tel que* e *s'évalue en* v*, alors aucun calcul intermédiaire ne produit une erreur de type.*

Donc, l'évaluation de l'expression est sûre pour l'interprète.

Remarque : v peut ne pas exister dans le cas où le calcul ne termine pas.

Remarque : Certains termes ne sont pas typables :
$\nvdash f : \texttt{function}\ (x : ?) \rightarrow f\ f$

Le système de typage : inférence de type

Principe : déduire des informations sur le type des termes à partir de leurs utilisations

Idée de base : attribuer une inconnue de type

- à chaque variable
- à chaque sous-terme

On en déduit un ensemble C de contraintes (=équations), qu'on résout par *unification*.

Le système de typage : inférence de type

Calcul des contraintes :

$$(\text{entier } i)_\alpha : \{\alpha = \texttt{int}\}$$

$$\texttt{true}_\alpha \text{ ou } \texttt{false}_\alpha : \{\alpha = \texttt{bool}\}$$

$$(e_\alpha \oplus e'_\beta)_\gamma : \{\alpha = \tau, \beta = \tau', \gamma = \tau''\} \text{ si } \oplus : \tau \to \tau' \to \tau''$$

$$(\texttt{function } x_\alpha \to e_\beta)_\gamma : \{\gamma = \alpha \to \beta\}$$

$$(e_\alpha\ e_\beta)_\gamma : \{\alpha = \beta \to \gamma\}$$

Exemple : $\texttt{function } f \to f\ (f\ 3)$

Annotations : $(\texttt{function } f_\alpha \to (f_\alpha\ (f_\alpha\ 3_\beta)_\gamma)_\delta)_\eta$

Contraintes : $\beta = \texttt{int}$, $\alpha = \beta \to \gamma$, $\alpha = \gamma \to \delta$, $\eta = \alpha \to \delta$

Résolution :

$\beta = \texttt{int}$, $\gamma = \delta$, $\gamma = \beta$, $\eta = (\texttt{int} \to \texttt{int}) \to \texttt{int}$

Le système de typage : inférence de type, unification

Données :

- un ensemble d'équations de termes de la forme $t_i = t'_i$
- chaque t_i et t'_i est un terme (type) contenant éventuellement des inconnues

Sortie :

- si elle existe, une substitution s (=fonction) associant à chaque variable une valeur (terme), de telle manière que $s(t_i) = s(t'_i)$ pour chaque équation $t_i = t'_i$
- erreur sinon

La substitution s calculée doit être la plus générale possible :

- un terme t est *plus général* qu'un autre t' s'il existe une substitution s telle que $s(t) = t'$;
- une substitution s est *plus générale* qu'une autre s' si $s(t)$ est plus général que $s'(t)$ pour tout terme t.

Le système de typage : inférence de type, unification

Un algorithme (Martelli, Montanari (1982))

- Si possible, il transforme un ensemble $G = \{t_1 = t'_1, \ldots, t_n = t'_n\}$ d'équations en un autre ensemble $\{x_1 = u_1, \ldots, x_m = u_m\}$ d'équations où les x_i sont des variables.
- Sinon, il calcule une erreur $\bot$.

$$G \cup \{t = t\} \rightsquigarrow G$$

$$G \cup \{f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)\} \rightsquigarrow G \cup \{t_1 = t'_1, \ldots, t_k = t'_k\}$$

$$G \cup \{f(t_1, \ldots, t_k) = g(t'_1, \ldots, t'_{k'})\} \rightsquigarrow \bot$$

si $f \neq g$ ou $k \neq k'$

$$G \cup \{f(t_1, \ldots, t_k) = x\} \rightsquigarrow G \cup \{x = f(t_1, \ldots, t_k)\}$$

si x est une variable

$$G \cup \{x = t\} \rightsquigarrow G[x/t] \cup \{x = t\}$$

si $x \notin vars(t)$ et $x \in vars(G)$

$$G \cup \{x = f(t_1, \ldots, t_k)\} \rightsquigarrow \bot$$

si $x \in vars(f(t_1, \ldots, t_k))$

Le système de typage : inférence de type, unification

Exemple :

$\texttt{function}\ x \to \texttt{function}\ y \to \texttt{function}\ z \to x\ z\ (y\ z)$

Annotation :

$$(\texttt{function}\ x_\alpha \to (\texttt{function}\ y_\beta \to (\texttt{function}\ z_\gamma \to ((x_\alpha\ z_\gamma)_\omega\ (y_\beta\ z_\gamma)_\delta)_\zeta)_\eta)_\psi)_\phi$$

Contraintes :

Applications :	Fonctions :
$a_1 : \beta = \gamma \to \delta$	$f_1 : \eta = \gamma \to \zeta$
$a_2 : \alpha = \gamma \to \omega$	$f_2 : \psi = \beta \to \eta$
$a_3 : \omega = \delta \to \zeta$	$f_3 : \phi = \alpha \to \psi$

Solution :

$\phi = (\gamma \to \delta \to \zeta) \to (\gamma \to \delta) \to \gamma \to \zeta$ $\qquad [f_3, f_2, f_1, a_1, a_2, a_3]$

Le système de typage : inférence de type, unification

Exemple :

$$\texttt{function}\ f \to (\texttt{function}\ x \to f\ x\ x)\ (\texttt{function}\ x \to f\ x\ x)$$

Annotation :

$$(\texttt{function}\ f_\alpha \to ((\texttt{function}\ x_\beta \to ((f_\alpha\ x_\beta)_\gamma\ x_\beta)_\delta)_\omega\ (\texttt{function}\ x_{\beta'} \to ((f_\alpha\ x_{\beta'})_{\gamma'}\ x_{\beta'})_{\delta'})_{\omega'})_\psi)_\phi$$

Contraintes et unification :

$$\alpha = \beta \to \gamma \qquad \gamma = \beta \to \delta \qquad \omega = \beta \to \delta$$

$$\alpha = \beta' \to \gamma' \qquad \gamma' = \beta' \to \delta' \qquad \omega' = \beta' \to \delta'$$

$$\omega = \omega' \to \psi \qquad \ldots \qquad \ldots$$

Le système de typage : inférence de type, unification

Exemple :

$$\texttt{function } f \to (\texttt{function } x \to f\ x\ x)\ (\texttt{function } x \to f\ x\ x)$$

Annotation :

$$(\texttt{function } f_\alpha \to ((\texttt{function } x_\beta \to ((f_\alpha\ x_\beta)_\gamma\ x_\beta)_\delta)_\omega\ (\texttt{function } x_{\beta'} \to ((f_\alpha\ x_{\beta'})_{\gamma'}\ x_{\beta'})_{\delta'})_{\omega'})_\psi)_\phi$$

Contraintes et unification :

$$\alpha = \beta \to \gamma \qquad \gamma = \beta \to \delta \qquad \omega = \beta \to \delta$$

$$\Big|\ \text{Unification : } \beta = \beta', \gamma = \gamma'$$

$$\alpha = \beta' \to \gamma' \qquad \gamma' = \beta' \to \delta' \qquad \omega' = \beta' \to \delta'$$

$$\omega = \omega' \to \psi \qquad \ldots \qquad \ldots$$

Le système de typage : inférence de type, unification

Exemple :

$$\texttt{function}\ f \to (\texttt{function}\ x \to f\ x\ x)\ (\texttt{function}\ x \to f\ x\ x)$$

Annotation :

$$(\texttt{function}\ f_\alpha \to ((\texttt{function}\ x_\beta \to ((f_\alpha\ x_\beta)_\gamma\ x_\beta)_\delta)_\omega\ (\texttt{function}\ x_{\beta'} \to ((f_\alpha\ x_{\beta'})_{\gamma'}\ x_{\beta'})_{\delta'})_{\omega'})_\psi)_\phi$$

Contraintes et unification :

$$\alpha = \beta \to \gamma \qquad \gamma = \beta \to \delta \qquad \omega = \beta \to \delta$$

$$\Big| \text{ Unification : } \beta = \beta', \gamma = \gamma' \qquad \Big| \text{ Unification : } \delta = \delta'$$

$$\alpha = \beta' \to \gamma' \qquad \gamma' = \beta' \to \delta' \qquad \omega' = \beta' \to \delta'$$

$$\omega = \omega' \to \psi \qquad \ldots \qquad \ldots$$

Le système de typage : inférence de type, unification

Exemple :

$$\texttt{function}\ f \to (\texttt{function}\ x \to f\ x\ x)\ (\texttt{function}\ x \to f\ x\ x)$$

Annotation :

$$(\texttt{function}\ f_\alpha \to ((\texttt{function}\ x_\beta \to ((f_\alpha\ x_\beta)_\gamma\ x_\beta)_\delta)_\omega\ (\texttt{function}\ x_{\beta'} \to ((f_\alpha\ x_{\beta'})_{\gamma'}\ x_{\beta'})_{\delta'})_{\omega'})_\psi)_\phi$$

Contraintes et unification :

$$\alpha = \beta \to \gamma \quad \Big| \text{ Unification : } \beta = \beta', \gamma = \gamma' \quad \alpha = \beta' \to \gamma'$$

$$\gamma = \beta \to \delta \quad \Big| \text{ Unification : } \delta = \delta' \quad \gamma' = \beta' \to \delta'$$

$$\omega = \beta \to \delta \quad \Big| \text{ donc } \omega = \omega' \quad \omega' = \beta' \to \delta'$$

$$\omega = \omega' \to \psi \qquad \ldots \qquad \ldots$$

Le système de typage : inférence de type, unification

Exemple :

$$\texttt{function}\ f \to (\texttt{function}\ x \to f\ x\ x)\ (\texttt{function}\ x \to f\ x\ x)$$

Annotation :

$$(\texttt{function}\ f_\alpha \to ((\texttt{function}\ x_\beta \to ((f_\alpha\ x_\beta)_\gamma\ x_\beta)_\delta)_\omega\ (\texttt{function}\ x_{\beta'} \to ((f_\alpha\ x_{\beta'})_{\gamma'}\ x_{\beta'})_{\delta'})_{\omega'})_\psi)_\phi$$

Contraintes et unification :

$$\alpha = \beta \to \gamma \quad \Big| \text{ Unification : } \beta = \beta', \gamma = \gamma' \quad \alpha = \beta' \to \gamma'$$

$$\gamma = \beta \to \delta \quad \Big| \text{ Unification : } \delta = \delta' \quad \gamma' = \beta' \to \delta'$$

$$\omega = \beta \to \delta \quad \Big| \text{ donc } \omega = \omega' \quad \omega' = \beta' \to \delta'$$

$$\omega = \omega' \to \psi \quad \ldots \quad \ldots$$

Erreur : type récursif !

Le système de typage : inférence de type, unification

Il existe d'autres algorithmes d'unification. Certains ont une complexité linéaire dans la taille de l'entrée.

Le filtrage est également à base d'unification (de motifs).

L'unification est également utilisée dans de nombreux autres langages de programmation, comme par exemple les langages de programmation par contraintes (ex : Prolog).

Étrangetés du système de typage

Il est possible d'écrire des fonctions dont le type de retour n'est pas dépendant des types des paramètres.

```
# let rec f1 x = f1 x;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "Erreur";;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [];;
val f3 : 'a -> 'b = <fun>
```

Ca n'est pas grave car l'interprète sait gérer ces cas particuliers :

- l'évaluation de `f1` ne termine jamais (et donc `f1` ne peut pas retourner une valeur de type « surprise ») ;
- le second cas est dû au typage des exceptions ;
- le troisième au polymorphisme des listes.

Les types polymorphes faibles

Les applications partielles peuvent générer des *types polymorphes faibles* dus à des caractéristiques fonctionnelles impures de OCaml, qui sortent du cadre du cours.

```
# let a = id id;;
val a : '_a -> '_a = <fun>
# a;;
- : '_a -> '_a = <fun>
# a 5;;
- : int = 5
# a;;
- : int -> int = <fun>
```

On s'en débarasse en ajoutant des paramètres (η-*expansion*)

```
# let b x = (id id) x;;
val b : 'a -> 'a = <fun>
# b 5;;
- : int = 5
# b 'e';;
- : char = 'e'
```

Le λ-calcul

Church $\approx$ 1930

Base théorique de la programmation fonctionnelle.

Très simple !

Calcul syntaxique sur des termes.

Soit V un ensemble de *variables*.
Les termes sont formés récursivement par

- $v \in V$ est un terme ;
- *Abstraction* : si $v \in V$, et T est un terme, alors $(\lambda v.T)$ aussi ;
  - la variable v est *liée* dans le terme $(\lambda v.T)$;
  - le terme $(\lambda v.T)$ est un λ-*terme*.
    Intuition :
    - un λ-terme $\lambda x.T$ est une fonction de x ;
    - T est le *corps* de la fonction.
- *Application* : si T, T' sont des termes, alors $(T\ T')$ aussi.
  Intuition : si T est une fonction alors $(T\ T')$ est l'application de T à son argument T'

Le λ-calcul : notations

Les parenthèses peuvent être enlevées pour alléger :

$\lambda x.(x\ (\lambda y.(\lambda z.\lambda x.(x\ y))\ y))$ devient $\lambda x.x\ (\lambda y.(\lambda z.\lambda x.x\ y)\ y)$

Les abstractions successives peuvent être écrites sous forme simplifiée :

$\lambda x.x\ (\lambda y.(\lambda z.\lambda x.x\ y)\ y)$ devient $\lambda x.x\ (\lambda yzx.x\ y)\ y)$

Le λ-calcul : α-conversion et β-réduction

Seulement deux règles de transformation des termes :

- α-*conversion* : renommage des variables pour éviter les collisions

$$\lambda x.T[x] \rightarrow \lambda y.T[x/y]$$

- β-*réduction* : application d'une fonction à son argument (= substitution du paramètre par l'argument dans le corps de la fonction)

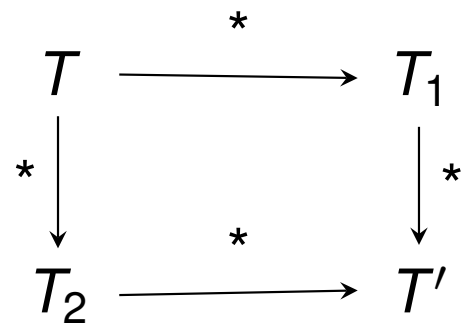
$$((\lambda x.T)\ T') \rightarrow T[x/T']$$

Un terme qui ne peut plus être (β-)réduit est en (β-)*forme normale*.
Deux termes T et T' sont β-*équivalents* ($T \equiv_\beta T'$) si $T \rightarrow_\beta^* T'$ ou $T' \rightarrow_\beta^* T$.

Le λ-calcul : confluence

Théorème

(Church-Rosser : confluence du λ-calcul)
Soit T un terme. S'il existe deux suites de β-réductions qui réduisent T en respectivement T_1 et T_2, alors il existe un terme T' et deux suites de β-réductions qui réduisent respectivement T_1 et T_2 en T'.



Corollaire

Si un terme T admet une forme normale, alors elle est unique.

Donc, peu importe l'ordre des β-réductions, le résultat final est toujours le même (aux boucles près !).

Le λ-calcul : codage des entiers

On code les entiers par le nombre d'applications d'une fonction f :

zero	$\lambda fx.x$
un	$\lambda fx.f\ x$
deux	$\lambda fx.f\ (f\ x)$
trois	$\lambda fx.f\ (f\ (f\ x))$

- ...

Calculer le successeur de n revient à appliquer f une fois de plus :

$$\texttt{Succ} : \lambda nfx.f\ (n\ f\ x)$$

L'addition de m et de n : $f^{n+m} = f^n f^m$

$$\begin{aligned}\texttt{Add} :& \lambda mnfx.m\ f\ (n\ f\ x)\\ & \lambda mnfx.n\ f\ (m\ f\ x)\\ & \lambda mn.m\ \texttt{Succ}\ n\\ & \lambda mn.n\ \texttt{Succ}\ m\end{aligned}$$

Le λ-calcul : les couples

On code un couple (x, y) et ses fonctions élémentaires par :

`Pair` $\lambda xyf.f\ x\ y$

`Fst` $\lambda c.c\ (\lambda uv.u)$

`Snd` $\lambda c.c\ (\lambda uv.v)$

On peut se servir des couples pour calculer.

Exemple

On définit $Decalage(m, n) = (n, n+1)$ par

$$Decalage : \lambda c.Pair\ (Snd\ c)\ (Succ\ (Snd\ c))$$

Alors $\phi(n+1) = (n, n+1)$ peut être définie par

$$\phi : \lambda n.n\ Decalage\ (Pair\ zero\ zero)$$

On en déduit `Pred` : $\lambda n.Fst\ (\phi\ n)$ `Sub` : $\lambda mn.n\ pred\ m$
Remarques : le prédecesseur de 0 est 0. $m - n = 0$ ssi $m \leq n$

Le λ-calcul : booléens et conditions

On code les booléens par :

true $\lambda xy.x$

false $\lambda xy.y$

et les opérations booléennes par :

And $\lambda xy.x\ y\ x$

Or $\lambda xy.x\ x\ y$

Not $\lambda x.x\ \mathit{false}\ \mathit{true}$

Ce qui permet de coder facilement un if/then/else :

$$\texttt{IfThenElse} : \lambda cxy.c\ x\ y$$

Et des tests

$$\texttt{IsZero} : \lambda nxy.n\ (\lambda z.y)\ x$$

Le λ-calcul : les listes, arbres, . . .

Les listes peuvent être codées en utilisant les couples. Une liste l est soit :

- la liste vide `Nil` : $\lambda x.$`true`
  `IsNil` : $\lambda p.p\ (\lambda xy.$`false`$)$
- un couple (v, l') où v est le premier élément de la liste et l' le reste de l

Les couples de couples de couples. . .peuvent être utilisés pour coder les arbres binaires. . .

Et on peut aussi définir beaucoup d'autres structures !

Le λ-calcul : codage des fonctions récursives

Par utilisation de la notion de *point fixe* : le terme P est un *point fixe* du terme T si $T\ P \equiv_\beta P$.

Théorème

Tout terme T a un point fixe.

Démonstration.

Soient $A = \lambda xy.y\ (x\ x\ y)$ et $\Theta = A\ A$. Alors $P = \Theta\ T$ est un point fixe de T car

$$\begin{aligned} P &= \Theta\ T \\ &= A\ A\ T \\ &= (\lambda xy.y\ (x\ x\ y))\ A\ T \\ &\rightarrow_\beta T\ (A\ A\ T) \\ &= T\ P \end{aligned}$$

□

Θ est appelé *opérateur de point fixe*.

Le λ-calcul : codage des fonctions récursives

$\texttt{Fact}$:
$\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \texttt{un}\ (\texttt{Mult}\ n\ (\texttt{Fact}\ (\texttt{Sub}\ n\ \texttt{un})))$

Soit $F =$
$\lambda f.\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \texttt{un}\ (\texttt{Mult}\ n\ (f\ (\texttt{Sub}\ n\ \texttt{un})))$

On cherche un terme $\texttt{Fact}$ vérifiant $\texttt{Fact} = F\ \texttt{Fact}$ (point fixe de F).

On a $\texttt{Fact} = \Theta\ F$

Ce principe s'applique à toute fonction récursive.

Le λ-calcul : codage des fonctions récursives

Remarque : il existe une infinité d'opérateurs de point fixe.

Exemple : l'opérateur (*normal-order*-) Y (Curry)

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

L'évaluation de Y dans un langage évaluant systématiquement tous les arguments part dans une boucle infinie.

L'opérateur suivant (*applicative-order* Y) est équivalent à Y mais s'évalue sans boucler infiniment :

$$Y' = \lambda f.(\lambda x.f\ (\lambda y.(x\ x)\ y))\ (\lambda x.f\ (\lambda y.(x\ x)\ y))$$

Le λ-calcul : codage des fonctions récursives

λ-calcul et programmation fonctionnelle sont équivalents

Les opérateurs de point fixe peuvent être utilisés pour implanter des fonctions récursives dans des langages L sans récursivité :

- si L a une stratégie d'évaluation paresseuse des paramètres, on peut utiliser Y ;
- sinon, on peut utiliser Y' (mais il est plus coûteux).

Programmations fonctionnelle et impérative sont équivalentes

- programme fonctionnel $\rightarrow$ impératif : facile
- programme impératif $\rightarrow$ fonctionnel : cf. cours de calculabilité (M1)

Plus sur le λ-calcul

Historiquement, le premier résultat d'indécidabilité est sur le λ-calcul :

Théorème

(Church) La question de l'équivalence de deux termes est indécidable.

λ-calcul typé = λ-calcul avec contraintes de type pour les termes.
Exemple : $\lambda x^{\tau \to \tau} y^{\tau}.x\ y$
Le λ-calcul typé

- a la plupart des bonnes propriétés du λ-calcul non typé ;
- a des propriétés supplémentaires importantes ;
- est plus proche de la logique mathématique et de la théorie des preuves (isomorphisme de Curry-Howard) ;
- est plus proche des langages de programmation typés ;
- est la base théorique d'OCaml !

`coreML` : un petit langage fonctionnel en OCaml

`coreML` est un petit langage d'expressions contenant :

- des entiers et des opérateurs arithmétiques ;
- des booléens et des opérateurs logiques ;
- des tests if/then/else ;
- des noms symboliques ;
- des fonctions unaires ;
- une construction `let` *expr* `in` *expr′*.

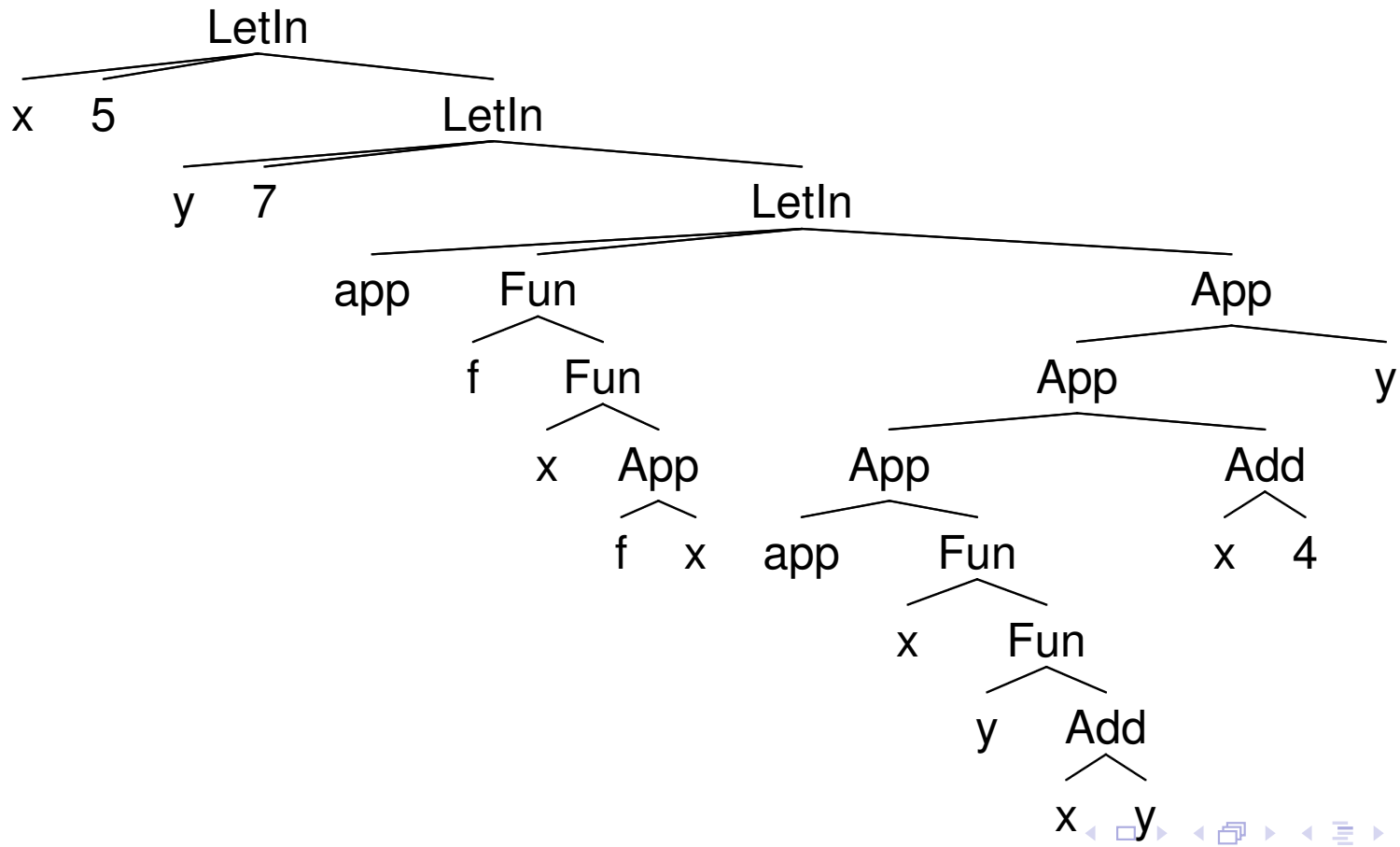
Exemple :

```
let x=5
in let y=7
   in let app = lambda f.lambda x.f x
      in ((app (lambda x.lambda y.x+y)) (x+4)) y
```

coreML : un petit langage fonctionnel en OCaml

Arbre de Syntaxe Abtrait (AST)

```
let x=5
in let y=7
   in let app = lambda f.lambda x.f x
      in ((app (lambda x.lambda y.x+y)) (x+4)) y
```



`coreML` : un petit langage fonctionnel en OCaml

Les expressions sont évaluées dans un *environnement* qui associe une valeur (expression) aux variables libres.

Exemples :

- la valeur de $x + 1$ dans un environnement qui à x associe 4 est 5 ;
- la valeur de `let` $x = 4$ `in` $x + 1$ dans un environnement vide est celle de $x + 1$ dans l'environnement qui à x associe 4 ;
- la valeur de $(\lambda x.x + 1)\ 4$ dans un environnement vide est celle de $x + 1$ dans l'environnement qui à x associe 4.

L'environnement peut être facilement représenté par une liste de couples (*nom*, *expr*) avec insertion en tête (mais ça n'est pas la représentation la plus efficace).

coreML : un petit langage fonctionnel en OCaml

Types pour l'implantation :

```
type
  expr =
    Int of int | Bool of bool
  | Add of expr*expr | Sub of expr*expr
  | Mul of expr*expr
  | Or of expr*expr | Not of expr | Eq of expr*expr
  | If of expr*expr*expr
  | LetIn of string*expr*expr
  | Fun of string*expr*env | App of expr*expr
  | Var of string
and
  env = (string*expr) list
;;

exception TypeError of string;;
exception NotDefined of string*env;;
```

coreML : un petit langage fonctionnel en OCaml

```
(* Cherche une valeur dans l'environnement *)
let find s env =
  let rec find_rec = function
      [] -> raise (NotDefined (s,env))
    | (x,v)::l when s=x -> v
    | _::l -> find_rec l
  in find_rec env
;;

(* Compare deux termes réduits *)
let eq e e' = (e=e')
;;
```

coreML : un petit langage fonctionnel en OCaml

Évaluation :

```
let rec add_env s e env = (s,e)::env
and eval env e =
  match e with
    Int _ as v -> v
  | Bool _ as v -> v
  | Add (e,e') -> begin
                    match (eval env e,eval env e') with
                      (Int i, Int i') -> Int (i+i')
                    | _ -> raise (TypeError "Add")
                    end
  ...
  | Eq (e,e') -> Bool (eq (eval env e) (eval env e'))
  | If (c,e,e') -> begin
                     match (eval env c) with
                       Bool b -> if b then eval env e
                                      else eval env e'
                     | _ -> raise (TypeError "If")
                     end
  ...
```

coreML : un petit langage fonctionnel en OCaml

Évaluation :

```
...
| Var s -> eval env (find s env)
| LetIn (s,e,e') -> let ve = eval env e
                    in eval (add_env s ve env) e'
| Fun (s,x,envfun) -> Fun (s,x,envfun@env)
| App (e,e') ->
        begin
        match (eval env e) with
           Fun (s,x,envfun) ->
              eval (add_env s (eval env e') envfun) x
        | _ -> raise (TypeError "App")
        end
;;
```

C'est fini !

coreML : un petit langage fonctionnel en OCaml

Exemples :

```
# let succExpr = Fun ("x", Add (Var "x",Int 1),[]);;
val succExpr : expr = Fun ("x", Add (Var "x", Int 1), []
# let e1Expr = App (succExpr, Int 10);;
val e1Expr : expr = ...
# eval [("x",Bool false)] e1Expr;;
- : expr = Int 11
```

Un peu plus difficile :

```
let x=5
in let y=7
   in let app = lambda f.lambda x.f x
      in ((app (lambda x.lambda y.x+y)) (x+4)) y
```

coreML : un petit langage fonctionnel en OCaml

```
# eval []
      (LetIn(
        "x",
        Int 5,
        LetIn(
            "y",
            Int 7,
            LetIn(
                "app",
                Fun("f",
                    Fun("x",
                        App(Var "f", Var "x"),[]
                       ),[]
                ),
                App(
                    App(
                        App(
                            Var "app",
                            Fun ("x",Fun("y",Add(Var "x",Var "y"),[]),[])
                           ),
                        Add(Var "x",Int 4)
                       ),
                    Var "y"
                )
            )
        )
      ))
;;
- : expr = Int 16
```

coreML : un petit langage fonctionnel en OCaml

Pour implanter les fonctions récursives il faut utiliser un opérateur de point fixe bien choisi pour ne pas boucler. Ici, « *applicative-order* Y »

$$Y' = \lambda f.(\lambda x.f\ (\lambda y.(x\ x)\ y))\ (\lambda x.f\ (\lambda y.(x\ x)\ y))$$

```
let applicative_order_Y =
  let yy =
    App(
        Var "f",
        Fun("y",
            App(
                App(Var "x",Var "x"),
                Var "y"
              ),[]
            )
      )
  in
  Fun ("f",
       App (
           Fun("x",
               yy,[]
              ),
           Fun("x",
               yy,[]
              )
           ),[]
      )
;;
```

coreML : un petit langage fonctionnel en OCaml

Il ne reste plus qu'à

- abstraire la définition récursive à évaluer

$$absfact = \lambda f.\lambda n.\texttt{IfThenElse}\,(\texttt{IsZero}\,n)\,\texttt{un}\,(\texttt{Mult}\,n\,(f\,(\texttt{Sub}\,n\,\texttt{un})))$$

- et à appliquer l'opérateur de point fixe

```
# let absfactExpr =
  Fun ("f",
      Fun ("n",
          If (Eq(Var "n",Int 0),
              Int 1,
              Mul(Var "n",App(Var "f",Sub(Var "n",Int 1)))
             ),[]),[]);;

val absfactExpr : expr = ...
# let factExpr = App(applicative_order_Y, absfactExpr);;
val factExpr : expr = ...
# eval [] (App(factExpr,Int 5));;
- : expr = Int 120
```

Unités de compilation

Les gros programmes peuvent être découpés en plusieurs unités de compilation (.ml).

Analogie avec le `C` :

	C	Byte-code	Natif
Code source	*.c	*.ml	*.ml
Fichiers d'en-tête	*.h	*.mli	*.mli
Fichiers objets	*.o	*.cmo	*.cmx2
Bibliothèque	*.a	*.cma	*.cmxa3
Fichiers exécutables	a.out	a.out	a.out

*.cmi : version compilée d'un .mli, obtenue par

```
$ ocamlc -c toto.mli
```

Modules

Permet l'isolation (la localisation) de types et fonctionnalités

```
# module Stack =
  struct
    type 'a stack = Stack of 'a list;;
    let push e = function Stack l -> Stack (e::l);;
    let pop = function Stack l -> Stack (List.tl l);;
    let top = function Stack l -> List.hd l;;
    let size = function Stack l -> List.length l;;
    let empty = Stack [];;
  end;;
module Stack :
  sig
    type 'a stack = Stack of 'a list
    val push : 'a -> 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val size : 'a stack -> int
    val empty : 'a stack
  end
```

Modules

Utilisation :

```
# let s = Stack.empty;;
val s : 'a Stack.stack = Stack.Stack []
# Stack.push 2 (Stack.push 1 s);;
- : int Stack.stack = Stack.Stack [2; 1]
```

ou bien, si l'utilisation de `Stack` est fréquente :

```
# open Stack;;
# let s = empty;;
val s : 'a Stack.stack = Stack []
# push 2 (push 1 s);;
- : int Stack.stack = Stack [2; 1]
```

ou bien, si l'utilisation de `Stack` est fréquente et locale :

```
# let open Stack in
    let p = push 2 (push 1 empty) in
      print_newline (print_int (size p));;
2
- : unit = ()
```

Interface de module

Il est possible de ne pas exposer la totalité d'un module. Par défaut, la totalité d'un module est exposé.

```
module Stack :
  sig
    type 'a stack;;
    val push : 'a -> 'a stack -> 'a stack;;
    val top : 'a stack -> 'a;;
    val size : 'a stack -> int;;
    val empty : 'a stack;;
  end
  =
  struct
    type 'a stack = Stack of 'a list;;
    let push e = function Stack l -> Stack (e::l);;
    let pop = function Stack l -> Stack (List.tl l);;
    let top = function Stack l -> List.hd l;;
    let size = function Stack l -> List.length l;;
    let empty = Stack [];;
  end;;
```

Interface de module

Il est possible de ne pas exposer la totalité d'un module. Par défaut, la totalité d'un module est exposé.

```
module Stack :
  sig
    type 'a stack
    val push : 'a -> 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val size : 'a stack -> int
    val empty : 'a stack
  end
# let s = Stack.push 2 (Stack.push 1 Stack.empty);;
val s : int Stack.stack = <abstr>
# Stack.size s;;
- : int = 2
# Stack.top s;;
- : int = 2
# Stack.pop s;;
Error: Unbound value Stack.pop
```

Modules et unités de compilation

Le module `Mystack` [1] peut être mis dans un fichier `mystack.ml`

- un nouveau module est automatiquement créé pour l'unité de compilation `mystack.ml` ;
- son nom `Mystack` est automatiquement calculé à partir du nom de l'unité de compilation `mystack.ml` ;
- l'interface du module peut être spécifiée dans `mystack.mli` ;
- si `mystack.mli` n'existe pas alors l'interface par défaut rend publique le contenu entier du module.

1. `Stack` a été renommé en `Mystack` pour éviter une collision avec le module `Stack` de la bibliothèque standard d'OCaml

Modules et unités de compilation

Fichier `mystack.ml` :

```
type 'a stack = Stack of 'a list;;

let push e = function Stack l -> Stack (e::l);;
let pop = function Stack l -> Stack (List.tl l);;
let top = function Stack l -> List.hd l;;
let size = function Stack l -> List.length l;;
let empty = Stack [];;
```

Fichier `mystack.mli` :

```
type 'a stack;;

val push : 'a  -> 'a stack -> 'a stack;;
val top : 'a stack -> 'a;;
val size : 'a stack -> int;;
val empty : 'a stack;;
```

Fichier `main.ml` :

```
let p = Mystack.push 2 (Mystack.push 1 Mystack.empty) in
    print_newline (print_int (Mystack.size p));;
```

Compilation de l'ensemble (attention à l'ordre) :

```
$ ocamlc -c mystack.mli              produit mystack.cmi
$ ocamlc -c mystack.ml               produit mystack.cmo
$ ocamlc -c main.ml                  produit main.cmo
$ ocamlc mystack.cmo main.cmo        produit a.out
```