Le Système Caml Light

Cours d'Informatique de

l'École Centrale d'Électronique

Ingénieur Deuxième Année

Sébastien Veigneau

Septembre 1997

Plan du Cours

1. Introduction
   - rappels ;
   - programmation impérative ;
   - programmation fonctionnelle ;
   - langages fonctionnels.
2. Le système Caml Light
   - types de base ;
   - expressions fonctionnelles ;
   - typage et polymorphisme ;
   - fonctionnelles d'itération.
3. Définition de types ;
4. Pattern matching et exceptions.

INTRODUCTION

Notion d'algorithme

- Procédé de calcul :

$$\boxed{\text{Entrée}} \xrightarrow{\text{Algorithme}} \boxed{\text{Sortie}}$$

- Chaque opération élémentaire est réalisable en **temps fini** ;

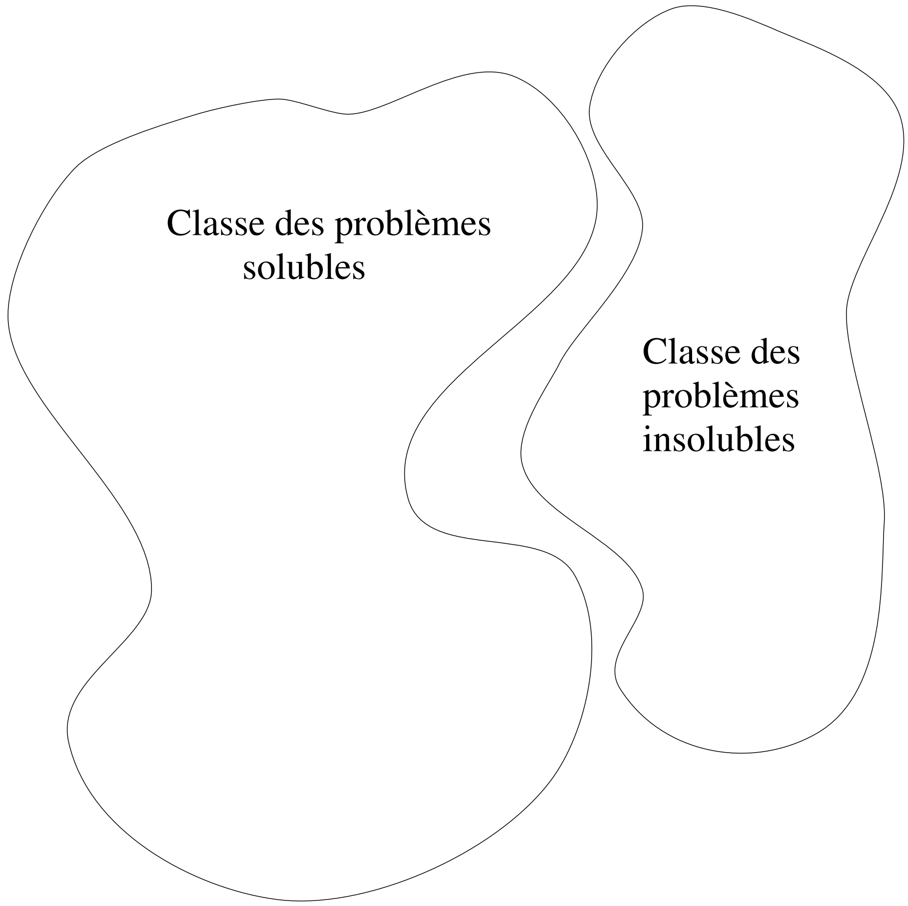
- Le nombre d'opérations est fini ;

- Donc, temps total d'exécution de l'algorithme **fini** ;

- Et la quantité de données manipulées l'est aussi.

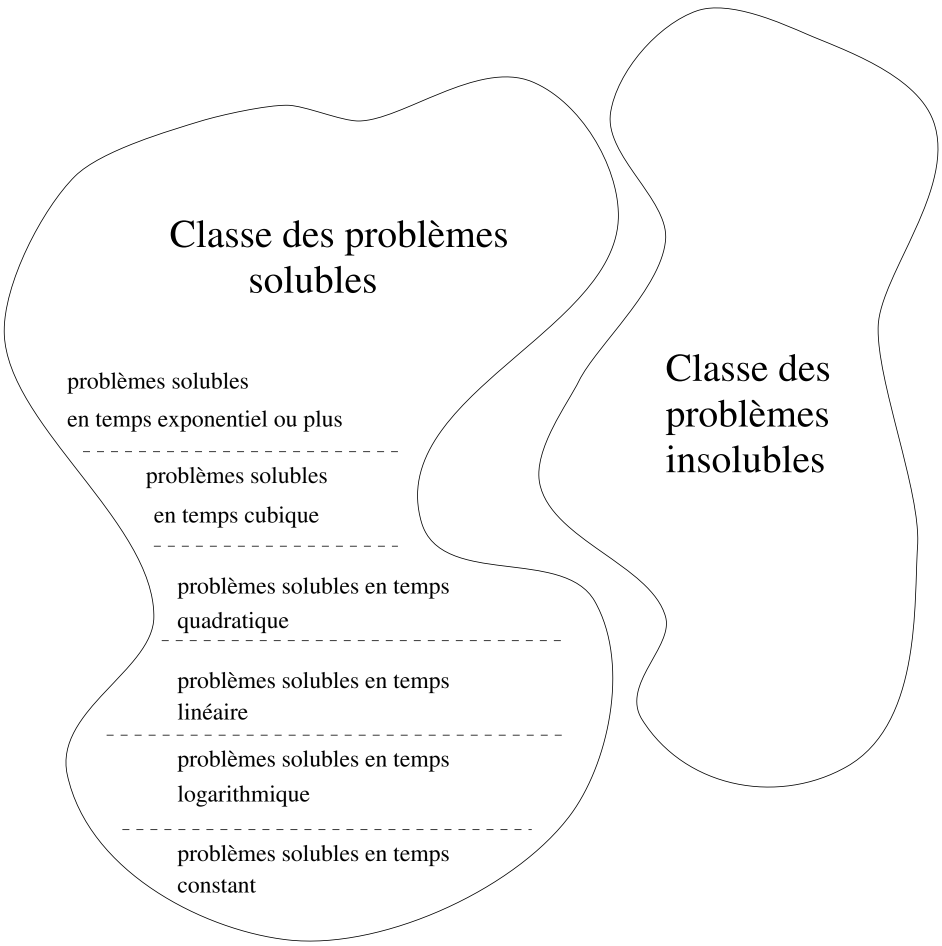
Problèmes (non) solubles

- Solution algorithmique ?



Classe des problèmes solubles

- **Classification des problèmes solubles** :



- **Complexité en temps**.

Programmation impérative

- Langages fondés sur deux concepts fondamentaux :
  - le concept de **variable** $\equiv$ **case mémoire** ;
  - le concept d'**instruction**.

- Modèle de machine très intuitif (Turing) :
  - processeur ;
  - mémoire.

- Machine de Turing :
  - ruban (cases) infini dans les deux sens ;
  - chaque contient 0 ou 1 ;
  - processeur (pointe sur une case du ruban) ;
  - programme (suite finie d'instructions).

Fonctions calculables

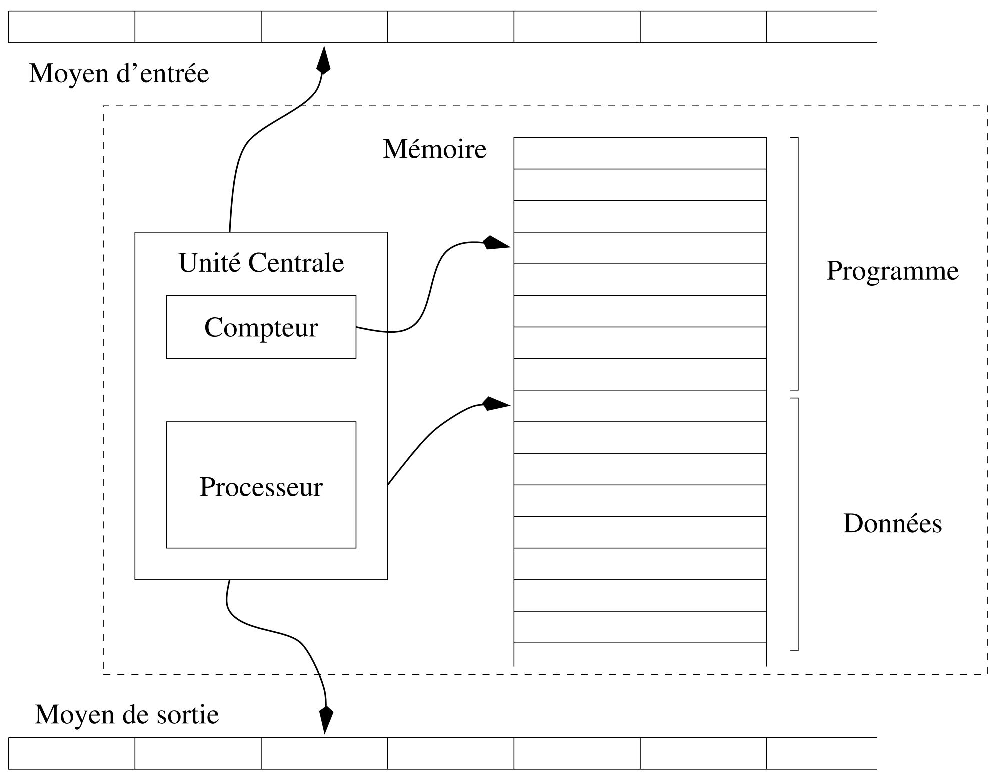
- Machine de Turing $\equiv$ ordinateur simplifié ;
- Modèle abstrait introduit pour répondre à :

Quelles sont les fonctions des entiers dans les entiers qui sont calculables ?

- Une fonction est calculable s'il existe un algorithme (fini) qui, étant donnés des arguments, permet de calculer un résultat.
- Formalisme des machines de Turing $\longrightarrow$ réponse à cette question : sont **calculables** les fonctions que l'on peut programmer au moyen d'une machine de Turing.

Modèle de machine abstraite

- Notion d'**algorithme** plongée dans un modèle de machine abstraite ;
- **RAM** (*Random Access Memory*) :
  - des moyens d'entrée/sortie ;
  - une mémoire ;
  - une unité centrale ;
  - un jeu d'instructions.
- Caractéristiques :
  1. accès direct de l'unité centrale à la mémoire ;
  2. mémoire infinie ;
  3. chaque case mémoire contient une donnée élémentaire ;
  4. chaque instruction simple s'exécute en temps constant.
- Objectif : comparer les performances d'algorithmes.



Machine abstraite RAM

Notion de variable

- **Algorithme** : décription d'une suite d'opérations à réaliser en fonction de l'état courant de la machine abstraite ;

- Présentation formelle d'un algorithme :

  langage algorithmique

- Manipulation de cases mémoire ;

- **Variable** : case mémoire de la machine abstraite.

Exemple

- Calcul du plus grand commun diviseur :

```
int pgcd(int m, int n)
{ while (!(m == n))
    if (m<n) n=n-m;
    else m=m-n;
  return m;
}

function pgcd(m,n:integer):integer;
begin
  while m<>n do
    if m<n then n:=n-m
    else m:=m-n;
  pgcd:=m
end
```

- Preuve de terminaison ?
- Preuve de correction ?

Preuve de programmes

- **Deux phases** :
  - preuve de correction ;
  - preuve de terminaison.
- **Preuve de correction** :
  - **preuve par récurrence** ;
  - **preuve par invariant** ;
  - problème difficile en général.
- **Preuve de terminaison** :
  - preuve par récurrence ;
  - considérations sur une variable qui ne cesse de diminuer par exemple, jusqu'à atteindre un cas de base ;
  - plus facile à montrer que la preuve de correction.

Langages fonctionnels

- Langages fonctionnels : ML, Caml, Caml Light ;

- Une seule notion fondamentale : **valeur** ;

- Calculs réalisés par des **fonctions** ;

- Fonctions :
  - objets comme les autres ;
  - argument : valeurs ;
  - retourne une valeur ;
  - très proche de la notion mathématique.

- Notion de **type**.

Langages de la famille ML

- Langages **fonctionnels**, où le calcul consiste à évaluer des expressions ;
- Langages **typés** où les types n'ont pas à être spécifiés dans les expressions ; ils sont synthétisés automatiquement par le compilateur ;
- Les types peuvent être **polymorphes** ;
- Il est possible d'y définir des types, notamment des types récursifs (listes, arbres. . .) très facilement et naturellement ;
- Possèdent un puissant mécanisme de programmation par cas : le **pattern matching** ;
- Dotés d'un mécanisme d'**exceptions** qui permet d'interrompre l'évaluation d'une expression ;
- Mécanismes de programmation impérative : Pascal peut être vu comme un sous-langage de ML.

Exemple

- Calcul du plus grand commun diviseur :

```
let rec pgcd(m, n) =
  if m=n then m
         else if m<n then pgcd(m,n-m)
                     else pgcd(m-n,n);;
```

- Fonction récursive ;
- Programmation récursive : style naturel ;
- Définition du PGCD = définition mathématique la plus simple de l'algorithme d'Euclide ;
- Fonctionnement : aucune intuition d'espace ni de temps ;
- **Raisonnement par récurrence**.

Historique

- **Alonzo Church** : λ-calcul (années 30)

  théorie des fonctions comme procédés de calcul $\rightarrow$ modèle important de la calculabilité.

- 1956 : Mac Carthy, **LISP**

  se veut une implémentation du λ-calcul, mais n'en est pas une.

- 1965 : ISWIM (If You See What I Mean)

  vrai précurseur de ML.

- 1978 : Milner, **ML** ;

- 1985 : Turner, **Miranda**.

LE SYSTÈME CAML LIGHT

Caml Light

- Lancement :

```
$ camllight
>       Caml Light version 0.73

#
```

- Exemples :

```
#1;;
- : int = 1

#2 + 3;;
- : int = 5

#function x -> x + 3;;
- : int -> int = <fun>

#(function x -> x + 3) 2;;
- : int = 5
```

Phrases et évaluation

- Une **phrase** est une suite de caractères qui se termine par **;;**
- Quatre types de phrases :
  - les expressions ;
  - les déclarations globales de valeurs ;
  - les déclarations de types ;
  - les déclarations d'exceptions.
- "**;;**" $\rightarrow$ phase de *synthèse* de type ;
- Affichage du type calculé de l'expression ;
- Compilation de l'expression ;
- Exécution et affichage du résultat ;

Les phrases entrées sont donc compilées et exécutées immédiatement

Expressions entières

- De type `int` :

```
#45;;
- : int = 45
```

- **Opérations de base**

  `+`, `-` (unaire et binaire), `*`, `/` et `mod`.

```
#(14 - 3*4) - (11 + 5) + 36;;
- : int = 22
```

- **Règle de typage** :

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 \texttt{+} e_2 : \texttt{int}}$$

- Fonction `prefix` :

```
#prefix +;;
- : int -> int -> int = <fun>
#(prefix +) 1 2;;
- : int = 3
```

Nombres réels

- De type `float` :

```
#45.0;;
- : float = 45.0
```

- **Opérations de base**

  `+.`, `-.` (unaire et binaire), `*.`, `/.`.

```
#(14.6 -. 3.1 *. 4.0) -. (11.2 +. 5.0) +. 36.3;;
- : float = 22.3
```

- **Autres opérations** : `sqrt` (racine carrée), `exp` (exponentielle), `log` (logarithme) et les opérations trigonométriques usuelles comme `sin`, `cos`, etc.
- **Attention au typage des expressions** :

```
#23 +. 4.5;;
Toplevel input:
>23 +. 4.5;;
>^^
This expression has type int,
but is used with type float.
```

Booléens

- De type `bool` : deux constantes `true` et `false` ;

```
#true;;
- : bool = true
```

- **Opérations de base** :

  conjonction, disjonction, négation

- **Règles de typage** :

$$\frac{e_1 : \texttt{bool} \quad e_2 : \texttt{bool}}{e_1 \;\texttt{\&}\; e_2 : \texttt{bool}} \qquad \frac{e_1 : \texttt{bool} \quad e_2 : \texttt{bool}}{e_1 \;\texttt{or}\; e_2 : \texttt{bool}}$$

$$\frac{e : \texttt{bool}}{\texttt{not}\; e : \texttt{bool}}$$

- Exemple :

```
#not (true & false);;
- : bool = true
```

Expression conditionnelle

- Pour tout type σ :

$$\frac{e : \texttt{bool} \quad e_1 : \sigma \quad e_2 : \sigma}{\texttt{if}\; e \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 : \sigma}$$

- Exemple :

```
#if (true or false) then 2+4 else -1;;
- : int = 6
```

- Toujours un `else` :

```
#if true then 1;;
Toplevel input:
>if true then 1;;
>             ^
This expression has type int,
but is used with type unit.
```

Prédicats de comparaison

- **Égalité** :

$$\frac{e_1 : \sigma \quad e_2 : \sigma}{e_1 \texttt{=} e_2 : \texttt{bool}}$$

- Exemples :

```
#true = false;;
- : bool = false
#1 = 2;;
- : bool = false
#prefix =;;
- : 'a -> 'a -> bool = <fun>
```

- **Ordre sur les entiers** : `<`, `>`, `<=` et `>=`

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 \texttt{<=} e_2 : \texttt{bool}}$$

- Exemples :

```
#2+3 < 7-4;;
- : bool = false
#2*3 >= 5;;
- : bool = true
```

Chaîne de caractères

- Suite de caractères en `"` ;
- De type `string` ;
- Par exemple :

```
#"exemple";;
- : string = "exemple"
```

- **Opérateur de concaténation** :

$$\frac{e_1 : \texttt{string} \quad e_2 : \texttt{string}}{e_1 \wedge e_2 : \texttt{string}}$$

- Par exemple :

```
#"exe" ^ "mple";;
- : string = "exemple"
#prefix ^;;
- : string -> string -> string = <fun>
```

Caractères

- De type `char` ;
- Par exemple :

```
#‘a‘;;
- : char = ‘a‘
#‘\097‘;;
- : char = ‘a‘
```

- Code *ASCII* → `char` :

```
#char_of_int 97;;
- : char = ‘a‘
```

- `char` → code *ASCII* :

```
#int_of_char ‘b‘;;
- : int = 98
```

Produits cartésiens

- Si $\sigma_1, \ldots, \sigma_n$ types ML, alors $\sigma_1 * \ \ldots \ * \sigma_n$ est un type ; c'est le type **produit cartésien** des types $\sigma_1,\ \ldots,\ \sigma_n$.
- **Règle de typage** :

$$\frac{e_1 : \sigma_1 \quad \ldots \quad e_n : \sigma_n}{(e_1, \ldots, e_n) : \sigma_1 * \cdots * \sigma_n}$$

- Par exemple :

```
#23, 56.0;;
- : int * float = 23, 56.0

#2, true, "exemple";;
- : int * bool * string = 2, true, "exemple"
```

- Test d'égalité :

```
#(1, true) = (2, true);;
- : bool = false
```

Le type unit

- Étrange car il n'existe qu'une seule valeur de type `unit` :

  `()`

- C'est ainsi que :

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

- Parfois utile pour des fonctions qui ont des "effets de bord" ;
- Plutôt contraire à la philosophie de la programmation fonctionnelle.

Déclarations globales

- Pour donner un nom à une valeur ;
- Identificateur ML :
  - suite de caractères ;
  - lettres, chiffres, `_` ou `'` ;
  - commençant par une lettre.
- Syntaxe :

  $$\texttt{let}\ x\ \texttt{=}\ e\texttt{;;}$$

  où x désigne un identificateur et `e` une expression ;

```
#let x = 3;;
x : int = 3
```

- Attention à la signification de :

```
#let x = 3;;
x : int = 3
#let x = 6;;
x : int = 6
```

- Déclaration préalable avant utilisation :

```
#3*z;;
Toplevel input:
>3*z;;
>  ^
The value identifier z is unbound.
#let z = 7 - 2;;
z : int = 5
#3*z;;
- : int = 15
```

Expressions fonctionnelles

- Très proches des définitions mathématiques :

$$x \mapsto 2x + 7$$

- Donner une expression dont la valeur est cette fonction ;
- Mot-clef `function` :

```
#function x -> 2*x + 7;;
- : int -> int = <fun>
```

- `x` est un paramètre formel (tout comme le x dans $x \mapsto 2x + 7$ que l'on appelle "variable muette") ;
- Application par juxtaposition :

```
#(function x -> 2*x + 7) 3;;
- : int = 13
```

- On peut donner un nom à une fonction :

```
#let f = function x -> 2*x + 7;;
```

```
f : int -> int = <fun>
#f 5;;
- : int = 17
#3 + f (2 + 4);;
- : int = 22
```

- On peut utiliser une fonction pour définir un autre objet :

```
#let g = function x -> (f x) + (f (2*x)) - x;;
g : int -> int = <fun>
#g 2;;
- : int = 24
```

- Il n'y a pas que `int -> int` :

```
#let h = function x -> (x<=7) or
                       ((x>1) & (x<>4));;
h : int -> bool = <fun>
```

ou encore :

```
#let g = function (x,y,z) ->
  ((if x then y else 1), if y>10 then "exemple"
                                 else z^"a");;
g : bool * int * string -> int * string = <fun>
```

- Une fonction peut rendre une fonction comme résultat :

```
#let f = function x -> (function y -> x + y);;
f : int -> int -> int = <fun>
#let g = f 2;;
g : int -> int = <fun>
#g 5;;
- : int = 7
#(f 3) 4;;
- : int = 7
#f 3 4;;
- : int = 7
```

- Si $\sigma_1, \ldots, \sigma_n$ sont des types ML, le type :

$$\sigma_1 \rightarrow (\sigma_2 \rightarrow (\ldots \rightarrow (\sigma_{n-1} \rightarrow \sigma_n) \ldots)) \quad ,$$

  est noté par convention :

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \sigma_{n-1} \rightarrow \sigma_n \quad .$$

- Si $e_1, \ldots, e_n$ sont des expressions ML, l'expression :

$$(\ldots((e_1\, e_2) e_3) \ldots e_n) \quad ,$$

  est notée par convention :

$$e_1\, e_2 \ldots e_n \quad .$$

Fonctionnelles

- Fonction qui prend une fonction en paramètre :

```
#let F = function f -> (f 3) + 5;;
F : (int -> int) -> int = <fun>
#F (function x -> x*(x - 1));;
- : int = 11
```

- Fonctionnelle prenant en argument une fonctionnelle :

```
#let F = function f ->
                    2*f (function x -> x + 1);;
F : ((int -> int) -> int) -> int = <fun>

#let G = function f ->
      (function x -> f (function y -> x + y));;
G : ((int -> int) -> 'a) -> int -> 'a = <fun>
```

- Équivalence syntaxe :

$$\texttt{let}\ f = \texttt{function}\ p_1 \to \texttt{function}\ p_2 \to \cdots$$
$$\cdots \to \texttt{function}\ p_n \to e \quad .$$

peut s'écrire :

$$\texttt{let}\ f\ p_1\ p_2\ \ldots\ p_n = e \quad .$$

- Exemple :

```
#let f = function x -> 2*x + 7;;
f : int -> int = <fun>
```

peut s'écrire :

```
#let f x = 2*x + 7;;
f : int -> int = <fun>
```

- Autre exemple :

```
#let g = function (x,y) ->
              function z -> x + (y 3) + z;;
g : int * (int -> int) -> int -> int = <fun>
```

s'écrit :

```
#let g (x,y) z = x + (y 3) + z;;
g : int * (int -> int) -> int -> int = <fun>
```

Fonctions à plusieurs arguments

- N'existent pas vraiment ;
- Moyens de simuler :
  - fonctions qui prennent des n-uplets en argument ;
  - fonctions qui rendent des fonctions en résultat.
- Par exemple :

```
#let f1 = function (x,y,z) -> (x + y)*z;;
f1 : int * int * int -> int = <fun>
#let f2 = function x -> function y ->
                function z -> (x + y)*z;;
f2 : int -> int -> int -> int = <fun>

#f1 (2, 1, 3);;
- : int = 9
#f2 2 1 3;;
- : int = 9
```

Fonctions récursives

- Exemple de la fonction factorielle :

```
#let fact n = if n=0 then 1 else n*fact(n - 1);;
Toplevel input:
>let fact n = if n=0 then 1 else n*fact(n - 1);;
>                                  ^^^^
The value identifier fact is unbound.
```

- Mot-clef : `rec` :

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

- Fonctions à plusieurs arguments :

```
#let rec add (x, y) =
  if x=0 then y
         else add(x - 1, y) + 1;;
add : int * int -> int = <fun>
```

Fonctions mutuellement récursives

- Mot-clef : `and` ;
- Exemples :

```
#let rec f n = if n=0 then 1
                      else g (n - 1)
 and     g n = if n=0 then 1
                      else f (n - 1) +
                           g (n - 1);;
#g 2;;
- : int = 3

#let rec pair n = if n=0 then true
                         else impair (n - 1)
 and   impair n = if n=0 then false
                         else pair (n - 1);;
pair : int -> bool = <fun>
impair : int -> bool = <fun>
#pair 25;;
- : bool = false
#impair 63;;
- : bool = true
```

Déclarations locales

- Calcul de $e : \mathbb{N} \to \mathbb{N}$ telle que $e(n) = 2^n$ avec la fonction somme :

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

- $T(n)$ nombre d'additions :

$$
\begin{aligned}
T(0) &= 0 \quad , \\
T(n+1) &= 2T(n) + 1 \quad ,
\end{aligned}
$$

donc $T(n) = 2^n - 1$;

- Utilisation d'une déclaration locale :

```
#let rec exp' n =
   if n=0 then 1
          else (let x = exp' (n - 1) in x + x);;
exp' : int -> int = <fun>
```

- Syntaxe générale :

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \quad ,$$

x identificateur et e, e' expressions ;

- Déclaration locale = expression ;
- Déclaration locale de valeurs récursives :

```
#let rec f n =
  if n=0 then 1
         else n*f (n - 1) in (f 3) + (f 4);;
- : int = 30
```

- Variante syntaxique :

$$e'\ \texttt{where}\ x\ \texttt{=}\ e \quad ;$$

- Exemple :

```
#(f 3) + (f 4) where
  rec f n = if n=0 then 1
                    else n*f (n - 1);;
- : int = 30
```

Typage et polymorphisme

- Détermination complète d'un type :

```
#function (x, y) -> if x then y + 1 else y*y;;
- : bool * int -> int = <fun>
```

- Contraintes insuffisantes :

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

- L'expression est **polymorphe**, de type :

$$\alpha \times \beta \rightarrow \alpha \quad ,$$

α, β "schémas de type" ;

- Application à n'importe quel type :

```
#f (true, 3);;
- : bool = true
#f (7, "exemple");;
- : int = 7
```

- Exemple de la composition de fonctions :

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

- **Currification** :

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

`add` version **currifiée** de `plus` (Haskell Curry) ;

```
#let curry f = function x ->
                 function y -> f (x,y);;
curry : ('a * 'b -> 'c) ->
                  'a -> 'b -> 'c = <fun>
#let uncurry f = function (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) ->
                   'a * 'b -> 'c = <fun>

#compose curry uncurry;;
- : ('_a -> '_b -> '_c) ->
               '_a -> '_b -> '_c = <fun>
```

Listes

- Suite d'éléments construits à partir de `[]` et `::` ;
- $[e_1; e_2; ...; e_n] \equiv e_1 :: (e_2 :: ... (e_n :: []) ...)$;
- Exemples :

```
#1::2::[];;
- : int list = [1; 2]
#[3;4;5];;
- : int list = [3; 4; 5]
#let f = function x -> x in [f; f];;
- : ('a -> 'a) list = [<fun>; <fun>]
#[];;
- : 'a list = []
```

- **Extraction d'éléments** :

```
#let est_vide l = if l=[] then true
                          else false;;
est_vide : 'a list -> bool = <fun>
#let tete = function x::_ -> x
            | _ -> raise (Failure "tete");;
tete : 'a list -> 'a = <fun>
```

```
#let queue = function _::l -> l
             |    _ -> raise (Failure "queue");;
queue : 'a list -> 'a list = <fun>
```

- **Opérations sur les listes** :

```
#let rec lg = function [] -> 0
                   | _::l -> 1 + (lg l);;
lg : 'a list -> int = <fun>

#let rec append = function
        [], l2 -> l2
 | (x::l1), l2 -> x::(append (l1, l2));;
append : 'a list * 'a list -> 'a list = <fun>

#[1;2] @ [3;4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

#let rec rev = function [] -> []
               | x::l -> (rev l) @ [x];;
rev : 'a list -> 'a list = <fun>

#rev [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```

- **Fonctionnelle map** :

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

#map (function x -> x + 1) [3;5;7;9];;
- : int list = [4; 6; 8; 10]

#let rec map f = function
                   [] -> []
               | x::l -> (f x)::(map f l);;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map lg [[3;4]; [5;6;7;8]; []; [2]];;
- : int list = [2; 4; 0; 1]

#let map f = map_rec where rec map_rec =
 function
     [] -> []
 | x::l -> (f x)::(map_rec l);;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map (fun x y -> x + y) [1; 2; 3];;
- : (int -> int) list = [<fun>; <fun>; <fun>]
```

Fonctionnelles d'itération sur les listes

- Fonction sur les listes $\equiv$ itération d'une fonction ;
- Schéma général : supposons donnés $a : \sigma$ et $f : \tau \rightarrow \sigma \rightarrow \sigma$. La fonction définie par itération sur les listes à partir de a et f est la fonction g de type $\tau\,\texttt{list} \rightarrow \sigma$ donnée par :

$$\begin{aligned} g(\texttt{[]}) &= a \quad , \\ g(x::l) &= f\,x\,(g(l)) \quad ; \end{aligned}$$

- Autrement dit, g est la fonction qui à la liste $[x_1; x_2; \ldots; x_n]$ associe :

$$f\,x_1\,(f\,x_2\,\ldots(f\,x_n\,a)\ldots) \quad ;$$

- **Fonctionnelle** `list_it` :

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

- Exemples :

```
#let sigma = list_it 0 prefix +;;
sigma : int list -> int = <fun>
#sigma [1; 2; 3; 4; 5];;
- : int = 15

#succ 3;;
- : int = 4
#let lg = list_it 0 (fun _ n -> succ n);;
lg : '_a list -> int = <fun>
#lg [1; 2; 3; 4; 5];;
- : int = 5

#let append l1 l2 =
             list_it l2 (fun x l -> x::l) l1;;
append : 'a list -> 'a list -> 'a list = <fun>
#append [1; 2] [5; 6];;
- : int list = [1; 2; 5; 6]

#let rev = list_it [] (fun x l -> l @ [x]);;
rev : '_a list -> '_a list = <fun>
#rev [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
```

```
#let map f = list_it [] (fun x m -> (f x)::m);;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Dans la définition de `list_it`, à chaque appel récursif, les paramètres `f` et `a` sont repassés inchangés ;
- Cela consomme inutilement de l'espace mémoire :

```
#let list_it a f = g where rec g = function
      [] -> a
 | x::l -> f x (g l);;
list_it : 'a -> ('b -> 'a -> 'a) ->
                  'b list -> 'a = <fun>
```

- Exemple du calcul des parties d'une liste :

```
#let cons x l = x::l;;
cons : 'a -> 'a list -> 'a list = <fun>
#let subsets =
   list_it [[]]
           (fun x s -> s @ (map (cons x) s));;
subsets : '_a list -> '_a list list = <fun>
#subsets [1; 2; 3];;
- : int list list = [[]; [3]; [2]; [2; 3]; [1];
                      [1; 3]; [1; 2]; [1; 2; 3]]
```

Les vecteurs

- Analogues aux tableaux des langages impératifs ;
- Indexés sur le type `int` ;
- Premier indice : `0` ;
- Pour chaque type σ, σ `vect` est le type des vecteurs d'éléments du type σ ;
- Notation :

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

- Création :

```
#make_vect;;
- : int -> 'a -> 'a vect = <fun>
#make_vect 4 (3, true);;
- : (int * bool) vect = [|3, true; 3, true;
                          3, true; 3, true|]
#vect_of_list;;
- : 'a list -> 'a vect = <fun>
```

```
#vect_of_list [3; 2; 8];;
- : int vect = [|3; 2; 8|]
```

- Longueur d'un vecteur :

```
#vect_length;;
- : 'a vect -> int = <fun>
#let t = [|1; 5; 2; 8|];;
t : int vect = [|1; 5; 2; 8|]
#vect_length t;;
- : int = 4
```

- Extraction d'un élément :

```
#t.(3);;
- : int = 8
#t.(0);;
- : int = 1
```

- Modification d'un élément :

```
#t.(1) <- 10;;
- : unit = ()
#t;;
- : int vect = [|1; 10; 2; 8|]
```

DÉFINITION DE TYPES

Introduction

- Possibilité de définir ses propres types de données ;
- Types **récursifs**, sans utiliser la notion intermédiaire de pointeur par laquelle on est obligé de passer en langage *Pascal* ou *C*.
- La gestion de la mémoire n'a donc pas à être faite par l'utilisateur ;
- Types **polymorphes**.

- Plan général :
  1. Types énumérés ;
  2. Somme de deux types ;
  3. Types récursifs.

Les types énumérés

- Exemple d'un type à trois éléments :

```
#type couleur = Bleu | Rouge | Vert;;
Type couleur defined.
```

- `Bleu`, `Vert` et `Rouge` sont des **constructeurs d'arité** 0 :

```
#Vert;;
- : couleur = Vert
```

- Utilisation du **pattern matching** :

```
#let val = function
     Bleu -> 1
  |  Vert -> 2
  | Rouge -> 3;;
val : couleur -> int = <fun>
#val Vert;;
- : int = 2
```

Somme de deux types

- Exemple :

```
#type int_ou_bool = I of int | B of bool;;
Type int_ou_bool defined.
```

- `I` et `B`, constructeurs d'arité 1 :

```
#I;;
- : int -> int_ou_bool = <fun>
#I 4;;
- : int_ou_bool = I 4
#B true;;
- : int_ou_bool = B true
```

- Utilisation du **pattern matching** :

```
#let f = function
   B x -> if x then 1 else 2
 | I x -> x + 4;;
f : int_ou_bool -> int = <fun>
#let f' = function
   B  true -> 1
```

```
  | B false -> 2
  | I     x -> x + 4;;
f' : int_ou_bool -> int = <fun>
```

- Un exemple plus évolué :

```
#type string_ou_intlist = S of string
                        | IL of int list;;
Type string_ou_intlist defined.
```

- Utilisation du **pattern matching** :

```
#let lg = function S str -> string_length str
                | IL lst -> list_length lst;;
lg : string_ou_intlist -> int = <fun>

#lg (S "exemple");;
- : int = 7
#lg (IL [1; 2; 5; 3]);;
- : int = 4
```

Schéma de type

- Possibilité de définir un **schéma de type** qui définit **une fois pour toutes** la somme de deux types quelconques ;
- Utilisation de variables de types pour identifier les deux types dont on fait la somme :

```
#type ('a, 'b) somme = G of 'a | D of 'b;;
Type somme defined.
```

- `G` et `D`, constructeurs polymorphes :

```
#G 1;;
- : (int, 'a) somme = G 1
#G true;;
- : (bool, 'a) somme = G true
#D [1; 2];;
- : ('a, int list) somme = D [1; 2]
```

- Utilisation du **pattern matching** :

```
#let gd = function G x -> true | D x -> false;;
gd : ('a, 'b) somme -> bool = <fun>
```

```
#gd (G true);;
- : bool = true
#gd (D "bonjour");;
- : bool = false
```

- Autre exemple :

```
#let f = function G x -> D x
                | D y -> G (y + 1);;
f : ('a, int) somme -> (int, 'a) somme = <fun>
#f (G "bonjour");;
- : (int, string) somme = D "bonjour"
#f (D 3);;
- : (int, 'a) somme = G 4
```

Types récursifs

- Un exemple très simple :

```
#type entier = Zero | Succ of entier;;
Type entier defined.
```

- `Zero` constructeur d'arité nulle ;
- `Succ` constructeur d'arité 1 ;
- Redéfinition des fonctions arithmétiques usuelles :

```
#let succ x = Succ x;;
succ : entier -> entier = <fun>
#let pred = function   Zero -> Zero
                     | Succ x -> x;;
pred : entier -> entier = <fun>
```

- Programmation généralement récursive :

```
#let rec add = function
       (Zero, x) -> x
 | ((Succ x), y) -> Succ (add (x, y));;
add : entier * entier -> entier = <fun>
```

```
#let rec soust = function
     (x, Zero) -> x
 | (x, Succ y) -> pred (soust (x, y));;
soust : entier * entier -> entier = <fun>
#let rec mult = function
       (Zero, x) -> Zero
 | ((Succ x), y) -> add (y, mult (x, y));;
mult : entier * entier -> entier = <fun>
```

- D'où intérêt à définir un schéma général de définition par itération sur les entiers : soit σ un type, soit $a : \sigma$ et soit $f : \sigma \to \sigma$. La fonction obtenue par itération de f à partir de a sur les entiers est la fonction g donnée par :

$$\begin{cases} g(0) &= a \\ g(S(n)) &= f(g(n)) \end{cases}$$

où S est le successeur :

```
#let entier_it f a = g where rec g = function
     Zero -> a
 | Succ x -> f (g x);;
entier_it : ('a -> 'a) -> 'a -> entier ->
            'a = <fun>
```

- Réécriture des fonctions précédentes :

```
#let add (x, y) = entier_it succ y x;;
add : entier * entier -> entier = <fun>
#let soust (x, y) = entier_it pred x y;;
soust : entier * entier -> entier = <fun>
#let mult (x, y) =
  entier_it (function z -> add (y, z)) Zero x;;
mult : entier * entier -> entier = <fun>
```

- **Autre exemple, redéfinition des listes d'entiers** :

```
#type liste_entiers = Nil
           | Cons of entier * liste_entiers;;
Type liste_entiers defined.
```

- Exemple d'utilisation :

```
#let rec somme = function
           Nil -> Zero
 | Cons (x, l) -> add (x, somme l);;
somme : liste_entiers -> entier = <fun>
```

- **On peut faire mieux** avec les `'a listes` :

```
#type 'a liste = Nil | Cons of 'a * ('a liste);;
Type liste defined.
```

- Exemple d'utilisation :

```
#let rec somme = function
           Nil -> Zero
 | Cons (x, l) -> add (x, somme l);;
somme : entier liste -> entier = <fun>
```

- Fonctionnelle d'itération :

```
#let liste_it f a = g where rec g = function
           Nil -> a
 | Cons (x, l) -> f x (g l);;
liste_it : ('a -> 'b -> 'b) -> 'b -> 'a liste ->
           'b = <fun>
```

- Véritable intérêt : **définition de nouveaux types** ;

- Exemple des **arbres binaires** :

```
#type 'a arbre = Vide
     | Noeud of 'a * ('a arbre) * ('a arbre);;
Type arbre defined.
```

- Exemples d'utilisation :

```
#let rec somme = function
                Vide -> 0
 | Noeud (x, t1, t2) -> x + (somme t1)
                         + (somme t2);;
somme : int arbre -> int = <fun>

#let max x y = if x<y then y else x;;
max : int -> int -> int = <fun>

#let rec ht = function Vide -> 0
 | Noeud (x, t1, t2) -> 1 + max (ht t1)
                               (ht t2);;
ht : 'a arbre -> int = <fun>

#let rec flat = function Vide -> []
 | Noeud (x, t1, t2) -> (flat t1) @
                        (x::flat t2);;
flat : 'a arbre -> 'a list = <fun>
```

- Fonctionnelle d'itération :

```
#let arbre_it a f = g where rec g = function
   Vide -> a
 | Noeud(x,t1,t2) -> f x (g t1) (g t2);;
arbre_it : 'a -> ('b -> 'a -> 'a -> 'a) ->
           'b arbre -> 'a = <fun>
```

- Réécriture des fonctions précédentes :

```
#let ht = arbre_it 0
            (fun x h1 h2 -> 1 + (max h1 h2));;
ht : 'a arbre -> int = <fun>
#let nbe = arbre_it 0
             (fun x n1 n2 -> 1 + n1 + n2);;
nbe : 'a arbre -> int = <fun>
#let flat = arbre_it []
              (fun x l1 l2 -> l1 @ (x::l2));;
flat : 'a arbre -> 'a list = <fun>
```

PATTERN MATCHING ET EXCEPTIONS

Constructions

Définition :

- toute constante est un motif : `bool`, `int`, `string`,..., constructeur d'arité 0 (`[]`) ;
- toute variable est un motif ;
- si C est un constructeur d'arité $n \geq 1$ et si $p_1, \ldots, p_n$ sont des motifs, alors $p = C(p_1, \ldots, p_n)$ est un motif.

Construction : `as`

```
#let f = function
                 [] -> []
 |             x::[] -> []
 | x::((y::m) as l) -> m@l;;
f : 'a list -> 'a list = <fun>
#f [1; 2; 3];;
- : int list = [3; 2; 3]
```

Motif filtrant : `_`

```
#let f = function
                  [] -> []
 |              _::[] -> []
 | _::((_::m) as l) -> m@l;;
f : 'a list -> 'a list = <fun>
```

Construction : `match` ... `with`

- si e, e_1,...,e_n sont des expressions et si p_1,...,p_n sont des motifs, alors :

  `match` e `with` p_1 `->` e_1 | ... | p_n `->` e_n

  est une expression, qui est équivalente à :

  `(function` p_1 `->` e_1 | ... | p_n `->` e_n`)` e

  Inversement :

  `function` p_1 `->` e_1 | ... | p_n `->` e_n

  peut s'écrire :

  `function x -> (match x with` p_1 `->` e_1 | ... | p_n `->` e_n`)`

Limitations

Matching not exhaustive :

```
#let f = function
   x::_ -> [x];;
Toplevel input:
>.......function
>   x::_ -> [x]..
Warning: this matching is not exhaustive.
f : 'a list -> 'a list = <fun>
```

Motifs : linéaires uniquement

```
#let eg = function
  (x,x) -> true
 |     _ -> false;;
Toplevel input:
>  (x,x) -> true
>     ^
The variable x is bound several times
in this pattern.
```

Exceptions

- Mécanisme permettant d'interrompre brutalement l'évaluation d'une expression :

```
#(2 + 3)*(5 - (raise (Failure "abandon !")) );;
Uncaught exception: Failure "abandon !"
```

- L'expression `Failure "abandon !"` est une exception :

```
- : exn = Failure "abandon !"
```

- `Failure` : constructeur d'exceptions ;

Fonction d'émission : `raise`

```
#raise;;
- : exn -> 'a = <fun>
```

- Exemple du calcul du n-ème élément d'une liste :

```
#let rec nth n = function
     [] -> raise (Failure "nth: liste vide")
```

```
 | x::l -> if n=1 then x else nth (n - 1) l;;
nth : int -> 'a list -> 'a = <fun>
#nth 3 [1; 2; 3; 4; 5; 6; 7];;
- : int = 3
#nth 10  [1; 2; 3; 4; 5; 6; 7];;
Uncaught exception: Failure "nth: liste vide"
```

Définitions : `exception`

```
#exception Erreur;;
Exception Erreur defined.
#Erreur;;
- : exn = Erreur
#raise Erreur;;
Uncaught exception: Erreur

#exception Trouve of int;;
Exception Trouve defined.
#Trouve 5;;
- : exn = Trouve 5

#exception P of int * bool;;
Exception P defined.
#exception F of (int->int)->int;;
```

```
Exception F defined.
```

Capture : `try`...`with`

- Soit e' l'expression suivante :

$$\texttt{try}\ e\ \texttt{with}\ q_1 \to e_1 \,|\, \ldots \,|\, q_n \to e_n$$

- q_i : motifs d'exception ;
- l'évaluation de e ne produit pas d'exception, mais se termine avec la valeur v. Valeur de $e' = v$;
- si l'évaluation de e produit une exception filtrée par l'un des motifs q_i. Si i = plus petit entier tel que q_i filtre l'exception, valeur de e' = valeur de e_i ;
- ou bien l'évaluation de e produit une exception qui non filtrée par aucun des q_i. L'évaluation de e' produit cette même exception.

```
#let rec ch p n = function
     [] -> raise Erreur
 | x::l -> if (p x) then raise (Trouve n)
                    else ch p (n + 1) l;;
ch : ('a -> bool) -> int ->
     'a list -> 'b = <fun>
```

- Exemple :

```
#let cherche p l = try (ch p 1 l) with
     Erreur -> raise (Failure "rien trouve")
 | Trouve n -> n;;
cherche : ('a -> bool) -> 'a list -> int = <fun>

#let pair n = (n mod 2)=0;;
pair : int -> bool = <fun>

#cherche pair [1; 5; 3; 4; 6; 7];;
- : int = 4

#cherche pair [1; 5; 3];;
Uncaught exception: Failure "rien trouve"
```