Le langage Caml

Pierre Weis
Pierre.Weis@inria.fr

INRIA – 26-27 octobre 1998

PLAN

1. Idées simples sur Caml et la programmation fonctionnelle.

2. Structures de données et de contrôle.

3. Un peu d'évaluation symbolique.

4. Un peu d'algorithmique.

Bibliographie:

"Le langage Caml",
Pierre Weis et Xavier Leroy, InterEditions 1993

"Manuel de Référence du langage Caml",
Xavier Leroy et Pierre Weis, InterEditions 1993

`http://pauillac.inria.fr/caml/index-fra.html`
`http://pauillac.inria.fr/caml/FAQ/index-fra.html`

Idées générales

Un langage facile et sûr

- Caml possède une sémantique simple et rigoureuse.
- Il intègre plusieurs styles de programmation:
  - la programmation fonctionnelle, basée sur la notion de calcul (et donc de fonction).
  - la programmation impérative classique, basée sur la notion d'effets (modification de variables, tableaux, boucles).

Aide au programmeur:

- La gestion de la mémoire est automatique et sûre.
- Mise au point des programmes: trace, printf, debuggeur avec retour arrière.
- Méthodologie de développement (passage à l'échelle):
  - petits programmes: système interactif.
  - gros programmes: compilateur et système de modules, makefiles.

Un langage sûr

Vérification **statique** des programmes *avant exécution*

- Syntaxe : vérification simple, élimination des non-sens.
- Sémantique : vérification complexe. Difficile à automatiser (en toute généralité, nécessite des démonstrations).
  - Typage : vérifie les contraintes d'appartenance aux ensembles.
    Automatique, sans déclarations dans les programmes.
  - Définition obligatoire des données complexes (types construits). Induit des contraintes d'appartenance à des ensembles.
  - Filtrage : analyse de cas sur les données. Détection automatique des cas oubliés et inutiles (test d'exhaustivité).

Vérifications **dynamique** des programmes *durant l'exécution*

- Bornes de tableaux et de chaînes.
- Exhaustivité du filtrage.
- Allocation et déallocation gérées par le système.

Notions de base

Programmes

Un programme Caml est un ensemble de définitions, suivi d'un calcul qui utilise ces définitions.

Phrases

Les programmes se décomposent en phrases. Une phrase se termine par deux ;

On utilise ici le système interactif qui donne le type des phrases entrées.

Définitions

Une définition est introduite par `let`.

`let` *nom* = *valeur*

```
let x = 1;;
x : int = 1

let s = "ok";;
s : string = "ok"

let s2 = "ok" ^ "ok";;
s2 : string = "okok"
```

Notions de base

Une définition n'est pas modifiable: le nom défini est définitivement lié à la valeur calculée lors de sa définition. C'est un synonyme pour la valeur calculée.

Portée statique: on ne peut pas utiliser un nom avant de l'avoir défini.

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

# pi / 2;;
>pi / 2;;
>^^
L'identificateur pi n'est pas défini.
```

est le symbole d'invite du système interactif.

Il sait parler français, si on lui demande poliment!

Notions de base

Définitions de fonctions

let f (x) = *valeur*

```
let double_string (s) = s ^ s;;
double_string : string -> string = <fun>
```

Remarque: il n'y a pas d'instruction `return`, on retourne le résultat de la dernière expression.

Définitions simplifiées

On supprime les parenthèses autour de l'argument.

let f x = *valeur*

Appels de fonctions

f (x)

Appels simplifiés

On supprime les parenthèses autour de l'argument.

f x

Attention: ne marche que pour les arguments simples, constantes ou variables.

f ($x + 1$)

Notions de base

Fonctions à plusieurs arguments

On écrit les arguments successifs:

`let` *f x y* = *valeur*

```
let concat s1 s2 = s1 ^ s2;;
concat : string -> string -> string = <fun>
```

Fonctions récursives

`let rec` *f x* = *valeur*

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

# fact 10;;
- : int = 3628800
```

Comparaisons: `<`, `>`, `=`, `<=`, `>=`, `<>`

Priorités des opérateurs

Mêmes conventions qu'en mathématiques.

- Opérations: $1 + 2 * 3 = 7$
- Opérations et fonctions:
  $f\ x + g\ y$ signifie $(f\ x) + (g\ y)$
  Vrai pour toutes les opérations:
  $f\ x\ op\ g\ y$ signifie $(f\ x)\ op\ (g\ y)$
- Fonctions et fonctions:
  $f\ g\ x$ signifie $f(g)(x)$,
  pas $f\ (g\ (x))$, qu'on écrit $f\ (g\ x)$.

Priorités des opérateurs

Exemples:

$$f\ x + g\ y \text{ signifie } (f(x)) + (g(y))$$

$$f\ x - 1 \text{ signifie } (f(x)) - 1, \text{ pas } f(x-1)$$

$$f\text{-1 signifie } f - 1, \text{ pas } f(-1)$$

Analogie: trigonométrie.

$sin\ x - 1$, $sin\ x + cos\ x$

Applications:

```
# fact 2 * fact 3;;
- : int = 12
# fact 3-1;;
- : int = 5

# fact -1;;
Entrée interactive:
>fact -1;;
>^^^^
Cette expression est de type int -> int,
mais est utilisée avec le type int.
```

Priorités induites par le typage:

```
if x >= 1 && y > x * x then ...
```

Chaînes de caractères

Accès aux caractères des chaînes : s.[i]

La numérotation commence à partir de 0. Le dernier caractère a pour numéro la longueur de la chaîne - **1**.

```
let nom = "toutou";;
nom : string = "toutou"

# nom.[1];;
- : char = `o`

# nom.[5];;
- : char = `u`

# string_length nom;;
- : int = 6
# nom.[string_length nom - 1];;
- : char = `u`
```

Sous-chaîne: `sub_string s index longueur`

```
# sub_string nom 0 3;;
- : string = "tou"
```

Palindromes

Version brute:

```
let rec palindrome s =
 if string_length s <= 1 then true else
 if s.[0] = s.[string_length s - 1] then
  palindrome (sub_string s 1 (string_length s - 2))
 else false;;
palindrome : string -> bool = <fun>

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

Remarque: les “tests” sont des prédicats.

```
# 1 > 2;;
- : bool = false
```

Version plus simple: on rend le résultat du test

```
let rec palindrome s =
 if string_length s <= 1 then true else
 s.[0] = s.[string_length s - 1] &&
 palindrome (sub_string s 1 (string_length s - 2));;
palindrome : string -> bool = <fun>
```

Opérateurs booléens

e1 `||` e2 signifie **if** e1 **then true else** e2

e1 `&&` e2 signifie **if** e1 **then** e2 **else false**

Précédences: comme en mathématiques
`||` est analogue à $+$ et `&&` à $*$
e1 `||` e2 `&&` e3 signifie donc e1 `||` (e2 `&&` e3)

Palindrome avec opérateurs booléens:

```
let rec palindrome s =
 string_length s <= 1 ||
 s.[0] = s.[string_length s - 1] &&
 palindrome (sub_string s 1 (string_length s - 2));;
palindrome : string -> bool = <fun>

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

Définitions locales

`let x = e1 in e2`

signifie que x vaut e1 pendant le calcul de e2 (et seulement pendant ce calcul).

Palindrome version définitive:

```
let rec palindrome s =
 let l = string_length s in
 l <= 1 || s.[0] = s.[l - 1] &&
 palindrome (sub_string s 1 (l - 2));;
palindrome : string -> bool = <fun>

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

“Presque pas d’exceptions ni de cas particuliers dans le langage”

une fonction est donc définissable localement, même à plusieurs arguments ou récursive ou les deux.

Boucle `for`

```
for i = e1 to e2 do
 actions
done
```

Définit i, puis exécute les actions,
avec i valant e1 puis e1 + 1, ..., e2.

Remarques

- On ne peut pas modifier la valeur de l'indice dans le corps de la boucle.
- La boucle est exécutée `e2 - e1 + 1` fois.
- L'indice n'est plus défini après la boucle.

Boucle `for`

Exemple: calculer l'image miroir d'une chaîne

```
let rev_string s =
 let l = string_length s in
 let c = create_string l in
 for i = 0 to l - 1 do
  c.[i] <- s.[l - 1 - i]
 done;
 c;;
rev_string : string -> string = <fun>
```

Modification en place des caractères d'une chaîne:

s .[i] <- nouveau caractère

Autre idée pour les palindromes:

```
let palindrome s =
 let r = rev_string s in
 r = s;;
palindrome : string -> bool = <fun>

let palindrome s = rev_string s = s;;
palindrome : string -> bool = <fun>

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

Programmation impérative: impression

Rien: `unit`

Le "rien", alias () est de type `unit`.

Résultat d'une boucle: ().

Impression: `print_*`, résultat ().

```
for i = 0 to 10 do
 print_int i
done;;
012345678910- : unit = ()
```

Impression avec format: `printf`

```
for i = 0 to 10 do
 printf__printf "%d " i
done;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Notation qualifiée des identificateurs

Notation qualifiée: on précise le nom du module où est défini l'identificateur, module__identificateur

```
# io__print_int;;
- : int -> unit = <fun>

# format__print_int;;
- : int -> unit = <fun>
```

Notation non qualifiée: on omet le module, c'est le système qui le trouve (par défaut c'est un ensemble raisonnable de modules d'identificateurs prédéfinis).

```
print_int;;
- : int -> unit = <fun>
```

Notation qualifiée des identificateurs

Ouverture d'un module: si l'on "ouvre" un module, le nom du module devient optionnel.

```
# printf;;
Entrée interactive:
>printf;;
>^^^^^^
L'identificateur printf n'est pas défini.

# #open "printf";;
# printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>

# printf "%d + %d = %d\n" 1 2 3;;
1 + 2 = 3
- : unit = ()

# #open "format";;
# printf "%d@." 1;;
1
- : unit = ()

# printf;;
- : ('a, formatter, unit) format -> 'a = <fun>
```

Programmation impérative: séquence

Séquence: e1 ; e2 ; ...; en
ou: `begin` e1 ; e2 ; ...; en `end`

Résultat: en

Remarques:

- e1 ; e2 signifie calculer e1, **jeter le résultat**, puis calculer e2.
- e1 ; e2 a toujours pour résultat la valeur de e2.
- si e1 ne produit pas d'effet e1 ; e2 est strictement équivalent à e2. Par exemple `1 + 2; 5` est équivalent à `5`.

```
for i = 10 downto 0 do
 print_int i; print_string " "
done;
print_newline();;
10 9 8 7 6 5 4 3 2 1 0
- : unit = ()
```

Programmation impérative: procédures

Procédure: fonction agissant par effets (résultat ou argument “rien”).

```
let count lim =
 for i = 0 to lim do printf "%d " i done;;
count : int -> unit = <fun>

# count 10;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Une procédure peut être récursive et comporter plusieurs paramètres.

```
let rec count i lim =
 if i > lim then () else
 begin
  printf "%d " i;
  count (i + 1) lim
 end;;
count : int -> int -> unit = <fun>

# count 5 10;;
5 6 7 8 9 10 - : unit = ()
```

Programmation impérative et conditionnelles

Séquence et `if then else`:

Une séquence dans les branches d'un `if then else` **doit** être entourée de `begin` et `end`.

Action conditionnelle: `if then` **sans** `else`

`if` *cond* `then` *action*

signifie `if` *cond* `then` *action* `else` ()

```
let rec count i lim =
 if i <= lim then
  begin
   printf "%d " i; count (i + 1) lim
  end;;
count : int -> int -> unit = <fun>

let rec rcount i lim =
 if i <= lim then
  begin
   rcount (i + 1) lim; printf "%d " i
  end;;
rcount : int -> int -> unit = <fun>

# rcount 0 10;;
10 9 8 7 6 5 4 3 2 1 0 - : unit = ()
# count 0 10;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Programmation impérative: boucles `while`

```
while condition do
 actions
done
```

Exécute les actions, tant que `condition` est vraie.

La condition doit être modifiée dans le corps de la boucle pour que la boucle termine: d'où la notion de valeurs modifiables ou mutables.

Programmation impérative: références

<u>Références : `'a ref`</u>

Idée: une case mémoire, ou un tableau à une case, ou un enregistrement à un champ.

- Création: **`ref`** *valeur initiale*.
- Accès: `!r` (contenu de r).
- Modification: r `:=` *nouvelle valeur* (affectation).

Les références servent en particulier à créer des variables au sens des langages algorithmiques classiques ("affectables").

Programmation impérative: références

```
# let i = ref 10;;
i : int ref = ref 10

# i := !i + 1;;
- : unit = ()

# !i;;
- : int = 11

# i;;
- : int ref = ref 11

# while !i >= 0 do
   printf "%d " !i;
   i := !i - 1
  done;
  printf "\n";;
11 10 9 8 7 6 5 4 3 2 1 0
- : unit = ()

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

Programmation impérative: tableaux

<u>Vecteurs : `'a vect`</u>

- Création: `make_vect`, [| e1 ; ...; en |]
- Accès: v .(index)
- Modification: v .(index) <- nouvelle valeur

```
let max_vect v =
 let l = vect_length v in
 if l = 0 then invalid_arg "max_vect" else
 let max = ref v.(0) in
 for i = 1 to l - 1 do
  let e = v.(i) in
  if e > !max then max := e
 done;
 !max;;
max_vect : 'a vect -> 'a = <fun>
```

Programmation fonctionnelle: portée des identificateurs

On ne peut utiliser un nom qu'après l'avoir défini.

Une fois défini, un nom ne change plus de valeur.

```
let pi = 3.14;;
pi : float = 3.14

let aire rayon = pi *. rayon *. rayon;;
aire : float -> float = <fun>

# aire 4.0;;
- : float = 50.24
```

Si l'on redéfinit l'identificateur `pi`, rien ne change

```
let pi = 6.28;;
pi : float = 6.28

# aire 4.0;;
  - : float = 50.24
```

Nécessaire:

```
let pi = "$\pi$";;
pi : string = "$\pi$"

# aire 4.0;;
  - : float = 50.24
```

Rassurant: "voir + haut" et "c'est dans la boîte".

Comme en Mathématiques: **Portée statique**.

Attention: portée différent de durée de vie.

Programmation fonctionnelle: portée des identificateurs

Écueil: redéfinition n'est pas modification (système interactif).

```
let pred x = x + 1;;
pred : int -> int = <fun>

let rec fact x =
 if x <= 1 then 1 else x * fact (pred x);;
fact : int -> int = <fun>

# fact 2;;
# Interrupted.

let pred x = x - 1;;
pred : int -> int = <fun>

# fact 2;;
# Interrupted.

let rec fact x =
 if x <= 1 then 1 else x * fact (pred x);;
fact : int -> int = <fun>

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

Portée des identificateurs

usage des références

Comme variables globales:

```
let count = ref 0;;
count : int ref = ref 0

let gen_sym () =
  count := !count + 1;
  "x" ^ string_of_int !count;;
gen_sym : unit -> string = <fun>

# gen_sym ();;
- : string = "x1"
# gen_sym ();;
- : string = "x2"
```

Comme variables locales temporaires:
(en C, variables automatiques)

```
let rcount lim =
 let i = ref lim in
 while !i >= 0 do
  printf "%d " !i;
  i := !i - 1
 done;
 printf "\n";;
rcount : int -> unit = <fun>

# rcount 10;;
10 9 8 7 6 5 4 3 2 1 0
- : unit = ()
```

Portée des identificateurs

usage des références

Comme valeurs ordinaires (arguments, résultats):

```
let incr_ref r i =
 r := !r + i;;
incr_ref : int ref -> int -> unit = <fun>

# let i = ref (-1);;
i : int ref = ref -1

# incr_ref i 2;;
- : unit = ()
# !i;;
- : int = 1
```

Comme variables locales rémanentes:
(en C, variables statiques)

```
let gen_sym =
 let count = ref 0 in
 let make_symbole () =
  count := !count + 1;
  "x" ^ string_of_int !count in
 make_symbole;;
gen_sym : unit -> string = <fun>
```

Programmation fonctionnelle: valeurs de premières classes

Les valeurs en Caml ont toutes le même statut.

On peut les traiter comme des entiers:

- les passer en argument
- les rendre en résultat
- les mettre dans des structures de données

C'est vrai en particulier pour les fonctions:

```
let succ x = x + 1;;
succ : int -> int = <fun>

# succ;;
- : int -> int = <fun>
```

On peut construire directement des fonctions, avec la construction:

`function` arg -> expression

```
# function x -> x - 1;;
- : int -> int = <fun>

let pred = function x -> x - 1;;
pred : int -> int = <fun>
```

`let f x = e` signifie `let f = function x -> e`

Programmation fonctionnelle

Fonction en résultat

Inutile de définir `make_symbole`: on renvoie une fonction!

```
let gen_sym =
 let count = ref 0 in
 function () ->
  count := !count + 1;
  "x" ^ string_of_int !count;;
gen_sym : unit -> string = <fun>
```

Fonction en argument

Sert à généraliser un algorithme.

```
let max comparaison x y =
 if comparaison x y then x else y;;
```

Fonction dans les structures de données

Sert à paramètrer un algorithme. Parfum d'orienté objet.

Programmation fonctionnelle: listes

Définition: `'a list`

- Liste vide: `[]`
- Constructeur (infixe): `::`
- Citation: liste des éléments, séparés par des `;`, entre `[` et `]`

```
# let l1 = [1; 2; 3];;
l1 : int list = [1; 2; 3]

# 0 :: l1;;
- : int list = [0; 1; 2; 3]

# let l2 = "toutou" :: "à" :: "sa" :: "mémère" :: [];;
l2 : string list = ["toutou"; "à"; "sa"; "mémère"]
```

Programmation fonctionnelle: listes

Accès: filtrage

```
match e with
| [] -> cas vide
| x :: rest -> cas non vide
```

Le “cas non vide” est celui de la liste formée de `x` et `rest`.

Exemple:

```
let rec print_string_list l =
 match l with
 | [] -> ()
 | x :: rest -> printf "%s " x; print_string_list l;;
print_string_list : string list -> unit = <fun>

# print_string_list ("Le" :: l2);;
Le toutou à sa mémère - : unit = ()
```

Vocabulaire:

- clause: `| filtre -> expression`
- filtre: définit la condition sur la forme de l’argument du filtrage pour que la clause s’applique; définit aussi les variables de la clause.
- partie expression de la clause: évaluée après liaison des variables du filtre aux morceaux correspondants de l’argument.

Polymorphisme

Polymorphisme: valeurs polymorphes

La liste vide est une liste de n'importe quel type:

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

Polymorphisme: fonctions polymorphes

```
let tete l =
 match l with
 | [] -> failwith "tete"
 | x :: l -> x;;
tete : 'a list -> 'a = <fun>

let queue l =
 match l with
 | [] -> failwith "queue"
 | x :: l -> l;;
queue : 'a list -> 'a list = <fun>

let rec length l =
 match l with
 | [] -> 0
 | x :: l -> 1 + length l;;
length : 'a list -> int = <fun>
```

Polymorphisme: fonctionnelles

Grâce au polymorphisme on peut écrire des fonctions très générales:

```
let rec iter f l =
 match l with
 | [] -> ()
 | x :: rest -> f x; iter f rest;;
iter : ('a -> 'b) -> 'a list -> unit = <fun>

# iter (function elem -> printf "%s " elem) l2;;
toutou à sa mémère - : unit = ()
```

Abréviation (argument anonyme, ou généralisation des fonctions):

```
let f x =
 match x with
 Filtrage

let f = function
 Filtrage

let rec iter f = function
 | [] -> ()
 | x :: rest -> f x; iter f rest;;
iter : ('a -> 'b) -> 'a list -> unit = <fun>
```

Polymorphisme: fonctionnelles

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

Libre mélange de purement fonctionnel et impératif.

```
let upper_char c =
 if c < `a` then c else char_of_int (int_of_char c - 32);;
upper_char : char -> char = <fun>

let to_upper s =
 let l = string_length s in
 let c = create_string l in
 for i = 0 to l - 1 do
  c.[i] <- upper_char s.[i]
 done;
 c;;
to_upper : string -> string = <fun>

# map to_upper l2;;
- : string list = ["TOUTOU"; "À"; "SA"; "MÉMÈRE"]
```

Programmation fonctionnelle: filtrage

Mécanisme très général, non réservé aux listes.

```
let rec fact = function
 | 0 -> 1
 | 1 -> 1
 | x -> x * fact (x - 1);;
fact : int -> int = <fun>
```

Filtre “_”

```
let closing = function
| `(` -> `)`
| `[` -> `]`
| `{` -> `}`
| `<` -> `>`
| _ -> failwith "closing";;
closing : char -> char = <fun>
```

Filtres “ou”

```
let rec fact = function
 | 0 | 1 -> 1
 | x -> x * fact (x - 1);;
fact : int -> int = <fun>
```

Abréviation

`f (x) = match x with` filtrage peut s’écrire
`f = function` filtrage

Programmation fonctionnelle: filtrage

Gardes

```
let rec fact = function
 | x when x <= 1 -> 1
 | x -> x * fact (x - 1);;
fact : int -> int = <fun>
```

Filtres intervalles

```
let advance_to_non_alpha s start =
 let rec advance i lim =
  if i >= lim then lim else
  match s.[i] with
  | `a` .. `z` | `A` .. `Z`
  | `0` .. `9` | `'` | `_`
  | `é`|`à`|`è`|`ù`|`â`|`ê`
  | `î`|`ô`|`û`|`ë`|`ï`|`ü`|`ç`
  | `É`|`À`|`È`|`Ù`|`Â`|`Ê`|`Î`
  | `Ô`|`Û`|`À`|`Ï`|`Ü`|`Ç` ->
     advance (i + 1) lim
  | _ -> i in
 advance start (string_length s);;
advance_to_non_alpha : string -> int -> int = <fun>
```

Programmation fonctionnelle: filtrage

Filtres synonymes “as”

```
(* Advance to the kth closing parens,
   getting rid of balanced parens *)
let advance_to_closing opening closing k s start =
 let rec advance k i lim =
  if i >= lim then raise Not_found else
  if s.[i] = opening
   then advance (k + 1) (i + 1) lim else
  if s.[i] = closing then
   if k = 0 then i else advance (k - 1) (i + 1) lim
  else advance k (i + 1) lim in
 advance k start (string_length s);;
advance_to_closing :
 char -> char -> int -> string -> int -> int = <fun>

(* Find an ident after a $ sign. *)
let find_ident s start =
 match s.[start] with
   (* Parenthesized ident ? *)
 | `(` | `[` | `{` | `<` as c ->
    let new_start = start + 1 in
    let stop =
      advance_to_closing c (closing c) 0 s new_start in
    sub_string s new_start (stop - start - 1), stop + 1
   (* Regular ident *)
 | _ ->
    let stop = advance_to_non_alpha s (start + 1) in
    sub_string s start (stop - start), stop;;
find_ident : string -> int -> string * int = <fun>
```

Programmation fonctionnelle: filtrage

Filtres complexes.

```
(* Substitute $ident (or $(ident)) in s,
    according to the function f_subst. *)
#open "buffer";;
let substitute_buffer f_subst s buf =
 let lim = string_length s in
 let rec subst previous i =
  if i < lim then
   begin
    match s.[i] with
    | `$` as current when previous = `\\` ->
       output_char buf current;
       subst current (i + 1)
    | `$` ->
       let ident, next_i = find_ident s (i + 1) in
       output_string buf (f_subst ident);
       subst ` ` next_i
    | current when previous = `\\` ->
       output_char buf `\\`;
       output_char buf current;
       subst current (i + 1)
    | `\\` as current ->
       subst current (i + 1)
    | current ->
       output_char buf current;
       subst current (i + 1)
   end in
 subst ` ` 0;;
```

Programmation fonctionnelle: exceptions

- Lancement: `raise` exception
- Rattrapage: `try` calcul `with` filtrage
- Définition: `exception` nom `of` type

Gestion des cas d'erreurs

```
try open_in "/tmp/foo"
with sys__Sys_error s -> ...
```

Cas particuliers courants:

- échec d'une recherche: exception `Not_found`
- argument invalide: `invalid_arg : string -> 'a` déclenche l'exception `Invalid_argument` avec la chaîne d'explication fournie.
- échec d'un accès à une structure `Invalid_argument`
- échec d'un filtrage: `Match_failure`
- échec de l'évaluation: `failwith` déclenche l'exception `Failure`.
- sortie prématurée d'une boucle `Exit`.

Programmation impérative: exceptions

Sorties de boucles

```
let is_in s c =
 try
  for i = 0 to string_length s - 1 do
   if s.[i] = c then raise Exit
  done;
  false
 with
 | Exit -> true;;
is_in : string -> char -> bool = <fun>
```

Avec retour d'une valeur

```
exception Found of int;;
L'exception Found est définie.

let pos_char s c =
 try
  for i = 0 to string_length s - 1 do
   if s.[i] = c then raise (Found i)
  done;
  raise Not_found
 with
 | Found i -> i;;
pos_char : string -> char -> int = <fun>
```

Programmation impérative: exceptions

Sorties de fonctions (peu employé)

```
# exception Return of int;;
L'exception Return est définie.

let return i = raise (Return i);;
return : int -> 'a = <fun>

let pos_char s c =
 try
  for i = 0 to string_length s - 1 do
   if s.[i] = c then return i
  done;
  return (-1)
 with
 | Return i -> i;;
pos_char : string -> char -> int = <fun>
```

Programmation impérative: exceptions

Capture temporaire d'une exception: pas de restriction dans le filtrage des exceptions.

```
let iter f path s =
 try f s with
 | sys__Sys_error _ as x ->
   let rec do_in = function
   | [] -> raise x
   | p :: pts ->
     try f (filename__concat p s) with
     | sys__Sys_error _ -> do_in pts in
   do_in path;;
iter :
 (string -> 'a) -> string list -> string -> 'a = <fun>
```

Programmation fonctionnelle: Autres types de données

`http://caml.inria.fr/man-caml/`

n-uplets: (x, y)

Création: automatique en ouvrant les parenthèses et en écrivant la virgule.

Accès: filtrage

Fonction à plusieurs arguments "tuplifiées"

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

Fonctions à plusieurs résultats

```
let div_eucl a b = (a/b, a mod b);;
div_eucl : int -> int -> int * int = <fun>

# div_eucl 7 5;;
- : int * int = 1, 2
```

Programmation fonctionnelle: paires

Définition simultanées (`let` déstructurant)

```
let q, r = div_eucl 7 5;;
q : int = 1
r : int = 2
```

Partage de références

```
let gensym, reset_gensym =
 let count = ref 0 in
 (function () ->
   count := !count + 1;
   "x" ^ string_of_int !count),
 (function () ->
   count := 0);;
gensym : unit -> string = <fun>
reset_gensym : unit -> unit = <fun>
```

La référence `count` n'est connue que des deux fonctions qui définissent `gensym` et `reset_gensym`.

Programmation fonctionnelle: application partielle

```
let tag t s = "<" ^ t ^ ">" ^ s ^ "</" ^ t ^ ">";;
tag : string -> string -> string = <fun>

# tag "HTML" "Titre";;
- : string = "<HTML>Titre</HTML>"

let h1 = tag "H1";;
h1 : string -> string = <fun>

# h1 "Mon titre";;
- : string = "<H1>Mon titre</H1>"

let h2 = tag "H2"
and h3 = tag "H3";;
h2 : string -> string = <fun>
h3 : string -> string = <fun>
```

Permet de spécialiser les fonctions,

de fabriquer des fonctions qui connaissent leur environnement.

Programmation fonctionnelle: définitions de types

Types enregistrement

```
type personne =
 {nom : string;
  téléphone : string;
  adresse_électronique : string};;
```

Types énumérés

```
type couleur = | Bleu | Blanc | Rouge;;
```

Types sommes

```
type numbers = | Int of int | Float of float;;
```

Programmation fonctionnelle: définitions de types

Définition des valeurs

```
let toto =
 {nom = "Toto";
  téléphone = "5563";
  adresse_électronique = "pauillac.inria.fr"};;

let c = Rouge;;

let pi = Float 3.14;;
```

Programmation fonctionnelle: manipulation des valeurs

Accès aux enregistrements: notation en .

```
let nom p = p.nom;;
nom : personne -> string = <fun>
```

Le filtrage est généralisé aux enregistrements

```
let tel {nom = _; téléphone = t;
         adresse_électronique = _} = t;;
tel : personne -> string = <fun>
```

Les champs inutiles peuvent être omis

```
let malle {adresse_électronique = a} = a;;
malle : personne -> string = <fun>
```

Programmation fonctionnelle: manipulation des valeurs

Accès aux valeurs de types sommes: filtrage

```
let string_of_couleur = function
 | Bleu -> "Bleu"
 | Blanc -> "Blanc"
 | Rouge -> "Rouge";;

let somme n1 n2 =
 match n1, n2 with
 | Int i1, Int i2 -> Int (i1 + i2)
 | Float i1, Float i2 -> Float (i1 +. i2)
 | Float i1, Int i2 -> Float (i1 +. float i2)
 | Int i1, Float i2 -> Float (float i1 +. i2);;
```

Programmation fonctionnelle: autres types

Enregistrements à champs mutables

`mutable` nom du champ
record . label `<-` nouvelle valeur

Types abréviation

type nom == expression de type

Types abstraits

Déclarés dans l'interface des modules, sans leur définition.

Modules

Modules simples: deux fichiers.

Interface: fichier ".mli"
contient des déclarations de valeurs avec leur documentation.

```
value fact : int -> int;;
 (* [fact i] renvoie la factorielle de
    l'entier [i] supposé positif.
    Déclenche l'exception [Invalid_argument "fact"]
    si ce n'est pas le cas. *)
```

Implémentation: fichier ".ml"
contient des définitions conformes à la documentation!

```
let rec fact x =
 if x < 0 then invalid_arg "fact" else
 ...
```

Utilisation du compilateur

Le compilateur caml compile en bytecode portable.

Types de fichiers manipulés:

- fichiers sources
  - `.ml` implémentation,
  - `.mli` interface.
- fichiers objets
  - `.zo` pour les implémentations
  - `.zi` pour les interfaces.

On peut écrire des programmes sans faire de modules.

- camlc f.ml : crée un exécutable a.out.
- camlc -o foo f.ml : crée un exécutable foo.
- camlc -c f.ml : compile le fichier seulement.
- camlc -i f.ml : compile f.ml en imprimant une interface qui exporte tout ce que contient f.ml (camlc -i f.ml > f.mli).
- camlc -g : compile en mode “debug”.
  Permet d'imprimer les exceptions imprévues avec **`printexc__print main ()`**.
- camlc *.zo : édition de liens.

Utilisation du compilateur

La méthode normale de travail sur un gros programme est d'utiliser `make` sous `emacs`.

CTRL-C CTRL-C pour tout recompiler,

CTRL-X ‘ pour aller directement au fichier et à la ligne où s'est produite l'erreur ou l'alerte du compilateur.

Un schéma de makefile est donné dans la documentation du langage (Foire Aux Questions).

```
# Caml compilers definition
CAMLC = camlc
CAMLDEP = camldep
CAMLLEX = camllex
CAMLYACC = camlyacc

# Source files automatically generated,
# including lexers and parsers
GENERATED = lexer.ml lexer.mli parser.ml parser.mli

# Executable file to produce
EXEC = exc

# Caml sources
SOURCES = types.ml main.ml $(GENERATED)
LEXSOURCES = lexer.mll
YACCSOURCES = parser.mly

# Caml object files
OBJS = types.zo lexer.zo parser.zo main.zo

...
```

Utilisation des librairies

Pour l'enseignement la principale est la librairie graphique.

Incantation minimale pour avoir le graphique:

ouvrir la librairie, puis la fenêtre graphique.

```
#open "graphics";;
```

```
open_graph "";;
```

Sous Unix, appeler `camllight camlgraph`.

```
(* Construit un carré dont le coin
   inférieur gauche est au point courant. *)
let carre c color =
 set_color color;
 let x, y = current_point () in
 lineto (x + c) y;
 lineto (x + c) (y + c);
 lineto x (y + c);
 lineto x y;;
```

Utilisation des librairies

```
let rec get_color = function
 | 0 -> black
 | 1 -> white
 | 2 -> red
 | 3 -> blue
 | 4 -> green
 | 5 -> yellow
 | 6 -> magenta
 | 7 -> cyan
 | i -> get_color (i mod 8);;

let centre () =
 moveto (size_x() / 2) (size_y() / 2);;

(* Carrés concentriques *)
let carrés n =
 clear_graph ();
 set_line_width 10;
 centre ();
 for i = 0 to n - 1 do
  carre (10 * i) (get_color i)
 done;;
```

Calcul symbolique

Les expressions

```
type expression =
   | Var of string
   | Const of float
   | Sum of expression * expression
   | Neg of expression
   | Prod of expression * expression
   | Inv of expression;;
```

L'évaluation

```
let rec eval env exp =
 match exp with
 | Var x -> assoc x env
 | Const c -> c
 | Sum (e1, e2) -> eval env e1 +. eval env e2
 | Neg e -> -. (eval env e)
 | Prod (e1, e2) -> eval env e1 *. eval env e2
 | Inv e -> 1.0 /. (eval env e);;
```

La dérivation symbolique

```
let rec deriv exp dx =
 match exp with
  | Var x -> if x = dx then Const 1.0 else exp
  | Const c -> Const 0.0
  | Sum (e1, e2) -> Sum (deriv e1 dx, deriv e2 dx)
  | Neg e -> Neg (deriv e dx)
  | Prod (e1, e2) ->
     Sum (Prod (e1, deriv e2 dx), Prod (deriv e1 dx, e2))
  | Inv e ->
     Neg (Prod (deriv e dx, Inv (Prod (e, e))));;
```

L'impression simple

```
let rec print exp =
 match exp with
  | Var x -> print_string x
  | Const c -> print_float c
  | Sum (e1, Neg e2) ->
     print_string "("; print e1;
     print_string " - "; print e2; print_string ")"
  | Sum (e1, e2) ->
     print_string "("; print e1;
     print_string " + "; print e2; print_string ")"
  | Neg e -> print_string "-"; print e
  | Prod (e1, Inv e2) ->
     print_string "("; print e1;
     print_string " / "; print e2; print_string ")"
  | Prod (e1, e2) ->
     print_string "("; print e1;
     print_string " * "; print e2; print_string ")"
  | Inv e -> print_string "1 / "; print e;;

print (deriv (Prod (Var "x", Var "x")) "x");;
((x * 1.0) + (1.0 * x))
```

Entrée des données

Il est difficile d'entrer des données un tant soit peu complexes.

Entrer l'expression `(x + 1) / (2 * x - 1)` revient à taper sans erreurs:

```
Prod
 (Sum (Var "x", Const 1.0),
  Inv (Sum (Prod (Const 2.0, Var "x"), Neg (Const 1.0))))
```

On cherchera donc à automatiser ce processus: c'est l'analyse lexico-syntaxique.

Deux moyens:

- Spartiate: en utilisant l'analyseur syntaxique de Caml.
- Sophistiqué: en utilisant les `streams` paresseux de Caml.
- Hors programme: en utilisant les générateurs d'analyseurs lexico-syntaxiques Lex et Yacc (`camllex`, `camlyacc`).

Entrée des données simple, par Caml

On définit des opérateurs infixes correspondants aux constructeurs: les priorités usuelles sont automatiquement respectées.

```
let prefix ++ e1 e2 = Sum (e1, e2)
and prefix ** e1 e2 = Prod (e1, e2)
and prefix -- e1 e2 = Sum (e1, Neg e2)
and prefix // e1 e2 = Prod (e1, Inv e2);;
```

On définit une injection pour les constantes

```
let i n = Const (float_of_int n);;
```

Et une ou des variables

```
let x = Var "x";;
```

L'entrée de `(Sum (Var "x", Var "x"))` devient:

```
x ++ x;;
- : expression = Sum (Var "x", Var "x")
```

`(x + 1) / (2 * x - 1)` devient:

```
let e = (x ++ i 1) // (i 2 ** x -- i 1);;
e : expression =
 Prod
  (Sum (Var "x", Const 1.0),
   Inv (Sum (Prod (Const 2.0, Var "x"), Neg (Const 1.0))))
```

Entrée des données par analyse syntaxique

```
#open "genlex";;

let lexer = make_lexer ["("; ")"; "+"; "-"; "*"; "/"];;

let rec parse_expr = function
  | [< parse_mult e1; (parse_more_adds e1) e >] -> e
and parse_more_adds e1 = function
  | [< 'Kwd "+"; parse_mult e2;
       (parse_more_adds (Sum (e1, e2))) e >] -> e
  | [< 'Kwd "-"; parse_mult e2;
       (parse_more_adds (Sum (e1, Neg e2))) e >] -> e
  | [< >] -> e1
and parse_mult = function
  | [< parse_basic e1; (parse_more_mults e1) e >] -> e;
and parse_more_mults e1 = function
  | [< 'Kwd "*"; parse_basic e2;
       (parse_more_mults (Prod (e1, e2))) e >] -> e
  | [< 'Kwd "/"; parse_basic e2;
       (parse_more_mults (Prod (e1, Inv e2))) e >] -> e
  | [< >] -> e1
and parse_basic = function
  | [< 'Ident x >] -> Var x
  | [< 'Int i >] -> Const(float_of_int i)
  | [< 'Float f >] -> Const f
  | [< 'Kwd "("; parse_expr e; 'Kwd ")" >] -> e;;

let read_expr s =
 parse_expr (lexer (stream_of_string s));;
```

Impression avec précédences

```
let rec print_expr = function
  | Sum (e1, Neg e2) ->
     print_expr e1; print_string " - "; print_mult e2
  | Sum (e1, e2) ->
     print_expr e1; print_string " + "; print_expr e2
  | Neg e ->
     print_string "-"; print_mult e
  | e -> print_mult e

and print_mult = function
  | Prod (e1, Inv e2) ->
     print_mult e1; print_string " / "; print_basic e2
  | Prod (e1, e2) ->
     print_mult e1; print_string " * "; print_mult e2
  | Inv e ->
     print_string "1 / "; print_basic e
  | e -> print_basic e

and print_basic = function
  | Var x -> print_string x
  | Const f -> print_float f
  | e ->
     print_string "("; print_expr e; print_string ")";;
```

Et maintenant

```
let e = read_expr "(x + 1) / (2 * x - 1)";;
e : expression =
 Prod
  (Sum (Var "x", Const 1.0),
   Inv (Sum (Prod (Const 2.0, Var "x"), Neg (Const 1.0))))

eval [("x", 1.0)] e;;
- : float = 2.0

eval [("x", 1.0)] (deriv e "x");;
- : float = -3.0
```

Au choix des clients

Les circuits numériques

Algorithmique classique avec vecteurs et affectations,
+ fonctions comme données.

Long mais facile.

Les arbres binaires de recherche rouge/noir

Algorithmique fonctionnelle.

C'est tout le contraire: pas long mais pas facile.

L'un ou l'autre ou les deux ?

Les circuits numériques

Définition

```
type circuit =
  { in_arity: int;
    out_arity: int;
    behavior: bool vect -> bool vect };;
```

Portes élémentaires

```
let not_gate =
  { in_arity = 1; out_arity = 1;
    behavior =
     function a -> [| not a.(0) |] };;

let and_gate =
  { in_arity = 2; out_arity = 1;
     behavior =
      function a -> [| a.(0) && a.(1) |] };;

let or_gate =
  { in_arity = 2; out_arity = 1;
    behavior =
     function a -> [| a.(0) || a.(1) |] };;

let wires n =
  { in_arity = n; out_arity = n;
    behavior = function a -> a };;
```

Mise en série et en parallèle

```
let prefix ||| c1 c2 =
  if c1.out_arity <> c2.in_arity then
   failwith "|||: arity mismatch" else
  { in_arity = c1.in_arity;
    out_arity = c2.out_arity;
    behavior =
     function a -> c2.behavior (c1.behavior a) };;

let prefix -- c1 c2 =
  { in_arity = c1.in_arity + c2.in_arity;
    out_arity = c1.out_arity + c2.out_arity;
    behavior =
     function a ->
      concat_vect
       (c1.behavior
         (sub_vect a 0 c1.in_arity))
       (c2.behavior
         (sub_vect a c1.in_arity c2.in_arity)) };;
```

Connexions

```
let fanout =
  { in_arity = 1;
    out_arity = 2;
    behavior =
     function a -> [| a.(0); a.(0) |] };;

let cross n i j =
  { in_arity = n;
    out_arity = n;
    behavior =
     function input ->
      let output = copy_vect input in
      output.(i) <- input.(j);
      output.(j) <- input.(i);
      output };;
```

Additionneur

```
let xor_gate =
     (fanout -- fanout)
 ||| cross 4 1 3
 ||| (or_gate -- (and_gate ||| not_gate))
 ||| and_gate;;

let half_adder =
     (fanout -- fanout)
 ||| cross 4 1 3
 ||| (xor_gate -- and_gate);;

let adder =
     (half_adder -- wires 1)
 ||| cross 3 1 2
 ||| (half_adder -- wires 1)
 ||| (wires 1 -- or_gate);;
```

Tables de vérité

```
let print_state a =
  for i = 0 to vect_length a - 1 do
    if a.(i) then print_string "1 "
    else print_string "0 "
  done;;

let truth_table c =
  let input = make_vect c.in_arity false in
  let rec generate i =
    if i < c.in_arity then begin
      input.(i) <- false; generate (i+1);
      input.(i) <- true;  generate (i+1)
    end else begin
      print_state input;
      print_string "--> ";
      print_state (c.behavior input);
      print_newline()
    end in
  generate 0;;
```

Tables de vérité de l'additionneur

```
truth_table and_gate;;
0 0 --> 0
0 1 --> 0
1 0 --> 0
1 1 --> 1
- : unit = ()

truth_table adder;;
0 0 0 --> 0 0
0 0 1 --> 1 0
0 1 0 --> 1 0
0 1 1 --> 0 1
1 0 0 --> 1 0
1 0 1 --> 0 1
1 1 0 --> 0 1
1 1 1 --> 1 1
- : unit = ()
```

Algorithmique:
arbres binaires de recherche

```
type 'a tree =
| Vide
| N of 'a tree * 'a * 'a tree;;
```

Invariant: les valeurs aux noeuds croissent de gauche à droite.

Exemple: `N (Vide, 4, Vide), 6, N (Vide, 8, Vide)`

Recherche dans l'arbre:

```
let rec member x t =
  match t with
  | Vide -> false
  | N (l, y, r) ->
     if x < y then member x l else
     if x > y then member x r else
     true;;
```

Insertion dans un a.b.r.

```
let rec insert x t =
  match t with
  | Vide -> N (Vide, x, Vide)
  | N (l, y, r) ->
     if x < y then N (insert x l, y, r) else
     if x > y then N (l, y, insert x r) else
     t;;
```

Problème: ne garantit pas que l'arbre est équilibré.

Dans le pire des cas, `member` et `insert` sont en $O(n)$.

Algorithmique: arbres rouge noir

```
type 'a tree =
| Vide
| Rouge of 'a tree * 'a * 'a tree
| Noir of 'a tree * 'a * 'a tree;;
```

Invariants:

- Arbre de recherche: valeurs croissantes de gauche à droite
- NN: tous les chemins de la racine vers une feuille ont le même nombre de noeuds `Noir`
- RR: les fils (directs) d'un noeud `Rouge` ne sont pas `Rouge`

Ceci garantit que l'arbre est équilibré: tous les chemins ont une longueur $O(\log n)$.

(Chris Okasaki, *Functional data structures*, Cambridge University Press, 1998.)

Recherche dans les a.r.n.

Même algorithme que pour les arbres de recherche:

```
let rec member x t =
 match t with
 | Vide -> false
 | Rouge (a, y, b) ->
    if x < y then member x a else
    if x > y then member x b else
    true
 | Noir (a, y, b) ->
    if x < y then member x a else
    if x > y then member x b else
    true;;
```

Complexité: $O(\log n)$ car l'arbre est équilibré.

Insertion dans les a.r.n.

Stratégie:

- Préserver l'invariant NN à chaque étape (en conservant le même nombre de noeuds noirs sur tous les chemins)

- Utiliser des rotations pour faire remonter les violations de l'invariant RR vers la racine

- Éliminer la dernière violation RR en mettant la racine en `Noir`

```
let insert x set =
 let rec ins = function
 | Vide ->
    Rouge (Vide, x, Vide)
 | Rouge (a, y, b) as t ->
    if x < y then Rouge (ins a, y, b) else
    if x > y then Rouge (a, y, ins b) else
    t
 | Noir (a, y, b) as t ->
    if x < y then balance (Noir (ins a, y, b)) else
    if x > y then balance (Noir (a, y, ins b)) else
    t in
 match ins set with
 | Rouge (a, y, b) -> Noir (a, y, b)
 | t -> t;;
```

Rotations

Faire disparaître les chemins `Noir-Rouge-Rouge` par rotation.

```
let balance = function

 (* Rouge Rouge à gauche puis à gauche *)
 | Noir (Rouge (Rouge (a, x, b), y, c), z, d) ->
    Rouge (Noir (a, x, b), y, Noir (c, z, d))

 (* Rouge Rouge à gauche puis à droite *)
 | Noir (Rouge (a, x, Rouge (b, y, c)), z, d) ->
    Rouge (Noir (a, x, b), y, Noir (c, z, d))

 (* Rouge Rouge à droite puis à gauche *)
 | Noir (a, x, Rouge (Rouge (b, y, c), z, d)) ->
    Rouge (Noir (a, x, b), y, Noir (c, z, d))

 (* Rouge Rouge à droite puis à droite *)
 | Noir (a, x, Rouge (b, y, Rouge (c, z, d))) ->
    Rouge (Noir (a, x, b), y, Noir (c, z, d))

 (* Invariant RR déjà respecté *)
 | t -> t;;
```

Dessiner les arbres

```
#open "graphics";;
let rec draw x y width t =
  match t with
  | Vide -> ()
  | Rouge (a, n, b) -> draw_node a n b red
  | Noir (a, n, b) -> draw_node a n b black
and draw_node a n b color =
  let abs = x + width / 2 in
  draw_son abs y x (y - 40) (width / 2) a;
  draw_son abs y abs (y - 40) (width / 2) b;
  draw_label abs y n color
and draw_son x0 y0 x1 y1 width t =
  match t with
  | Vide -> ()
  | _ ->
      set_color green;
      moveto x0 y0;
      lineto (x1 + width / 2) y1;
      draw x1 y1 width t
and draw_label x y value color =
  let txt = string_of_int value in
  let (width, height) = text_size txt in
  let abs = x - width / 2 in
  set_color color;
  fill_rect (abs - 2) (y - 2) (width + 4) (height + 4);
  set_color white;
  moveto abs y;
  draw_string txt;;
let draw_tree t =
  draw 0 (size_y() - 20) (size_x()) t;;
```