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.

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:

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

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

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

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.

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: 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.

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.

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: 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 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: 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);;
```

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.