Principes des lang. de progr.
INE 11

Michel Mauny

ENSTA ParisTech

Prénom.Nom@ensta.fr

Langages de programmation
Introduction à Objective Caml

1. Préliminaires
   - Objectifs du cours
   - Organisation
2. Les langages de programmation
   - Les grandes familles
   - Techniques de mise en œuvre
3. Objective Caml
   - Opérations de base
   - Fonctions et filtrage
   - Polymorphisme et modularité

TP à partir de la semaine prochaine

Motivations 1/2

Au début était l'hexadécimal :

Langage machine MIPS R4000 (calcul du PCGD)

```
27bdffd0 afbf0014 0c1002a8 0c1002a8 afa2001c 8fa4001c 00401825 10820008 0064082a
10200003 10000002 00832023 00641823 1483fffa 0064082a 0c1002b2 8fbf0014 27bd0020
03e00008 00001025
```

Puis vint le mnémonique :

Assembleur MIPS R4000

```
      addiu   sp,sp,-32                b       C
      sw      ra,20(sp)                subu    a0,a0,v1
      jal     getint                   subu    v1,v1,a0
      jal     getint             C :   bne     a0,v1,a
      sw      v0,28(sp)                slt     at,v1,a0
      lw      a0,28(sp)          D :   jal     putint
      move    v1,v0                    lw      ra,20(sp)
      beq     a0,v0,D                  addiu   sp,sp,32
      slt     at,v1,a0                 jr      ra
  A : beq     at,zero,B                move    v0,zero
```

Motivations 2/2

Et enfin les langages de haut niveau :

PGCD en Pascal

```
program pgcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i <> j do
    if i > j then
      i := i-j
    else
      j := j-i;
  writeln(i)
end.
```

En Objective Caml

```
let rec pgcd(i, j) =
  if i = j then i else
  if i > j then pgcd(i-j, j)
  else pgcd(i, j-i)
;;

open Printf;;

let arg i =
  int_of_string(Sys.argv.(i)) in
let (m,n) = (arg 1, arg 2) in
let _ = printf "%i\n" (pgcd(m,n)) in
exit 0;;
```

Parenthèse : dans ce cours, on notera «$\leq$», «$\neq$», «$\rightarrow$», ..., ce que dans les vrais programmes on écrit «<=», «<>», «->».

Motivations 2/2

Et enfin les langages de haut niveau :

PGCD en Pascal

```
program pgcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i ≠ j do
    if i > j then
      i := i-j
    else
      j := j-i;
  writeln(i)
end.
```

En Objective Caml

```
let rec pgcd(i, j) =
  if i = j then i else
  if i > j then pgcd(i-j, j)
  else pgcd(i, j-i)
;;

open Printf;;

let arg i =
  int_of_string(Sys.argv.(i)) in
let (m,n) = (arg 1, arg 2) in
let _ = printf "%i\n" (pgcd(m,n)) in
exit 0;;
```


Objectifs du cours

Développer une culture «langage»

- comprendre les langages de programmation *en général*
- avoir une idée de ce qui se cache derrière tel ou tel aspect de langage
  - comment ça marche ?
  - est-ce efficace ?
  - ce langage est-il adapté à cette tâche ?

Comment ?

- quelques éléments théoriques (analyses de programmes, sémantique)
- outils de définition de langage
- réaliser une maquette de langage en Objective Caml

Ce cours n'est pas ...

Pas un cours de compilation

- génération de code, allocation de registres, optimisation, *etc.* sont hors du champ de ce cours

Pas une introduction à tous les langages

- procédures/fonctions, gestion de la mémoire, objets
- pas de programmation logique ou par contraintes, (probablement) pas de mécanismes objets compliqués, pas de programmation concurrente, ni temps-réel, ni ...

Pas de la théorie pour la théorie

- ce dont on a besoin pour comprendre et « maquetter », et prendre un peu de recul

Organisation 1/2

1. Tour d'horizon des techniques de mise en oeuvre, premiers pas en OCaml
2. OCaml, la suite
3. Analyse lexicale (expressions rationnelles)
4. Analyse syntaxique (grammaires algébriques)
5. Termes, filtrage, unification.
6. Sémantique dénotationnelle
7. Stratégies d'évaluation (le λ-calcul)
8. Sémantique opérationnelle, interprétation et compilation
9. Gestion automatique de la mémoire
10. Vérification et synthèse de types
11. Objets
12. Projets et soutenances

Principes

- Le cours est obligatoire et commence à l'heure
- Cours : 1h30, TP : 2h. Pause : 15 min.
- Mini-projet : en TP à partir du bloc 5/6

Organisation 2/2

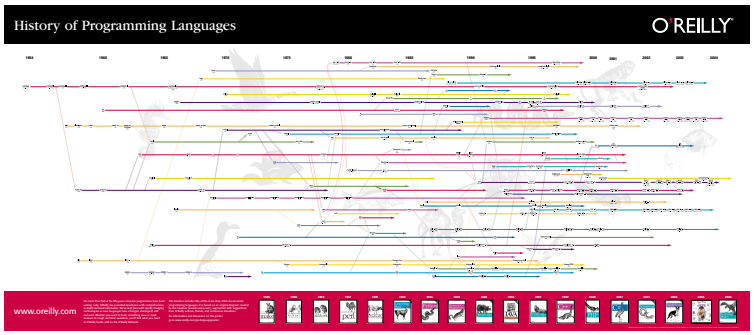
Documents :

- polycopié
- documentation Objective Caml

Page web du cours :

- `http://www.mauny.net/`
  → `Enseignement`
  → `Langages de programmation`

Le paysage



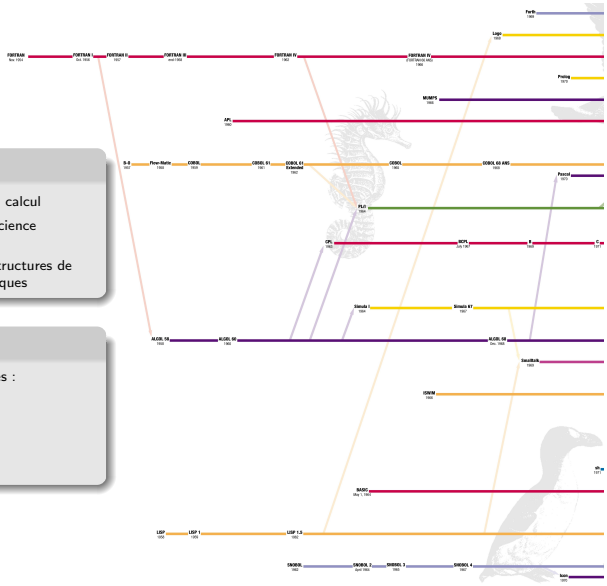
1960

- Fortran pour le calcul
- Algol pour la science informatique
- Lisp pour les structures de donnés dynamiques

1970

Influences et nouveautés :

- Prolog
- PL/1
- Smalltalk
- BASIC

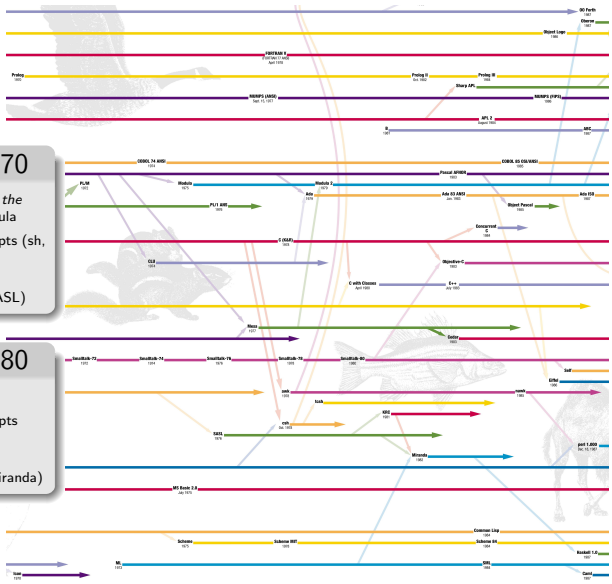


Tendances 1970

- *Programming in the large* : Clu, Modula
- Langages de scripts (sh, awk)
- Programmation fonctionnelle (SASL)

Tendances 1980

- Objets
- Langages de scripts (awk, rexx, perl)
- Programmation fonctionnelle (Miranda)



Tendances

- Java
- Langages de scripts (JavaScript, Ruby, PHP)
- Programmation fonctionnelle (Scheme, Caml, Haskell, SML)

Les grandes familles

Langages déclaratifs

- fonctionnels : Lisp/Scheme, ML/OCaml, Haskell, ...
- à flots de données : Sisal, SAC, ...
- logiques, à contraintes : Prolog, Excel (?), ...

Langages impératifs

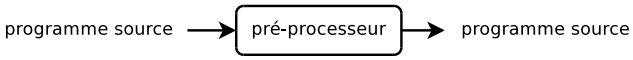
- classiques : Fortran, Pascal, Basic, C, ...
- à objets : Smalltalk, Eiffel, C++, Java, C#, Python, ...

Note : frontières floues

Techniques de mise en œuvre 1/4

1. Pré-traiter

- cpp, `m4`, macros des langages Lisp
- agissent sur le texte source des programmes
- sans «comprendre» le programme
- ni procéder à son exécution



Techniques de mise en œuvre 2/4

2. Compiler = traduire

- traduction d'un programme source en un programme «objet»
- donne du sens au programme : la traduction préserve le sens
- ne procède pas à l'exécution du programme (entrées non disponibles)

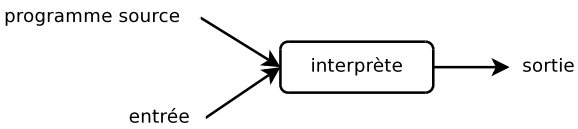
Attention

- «source» $\not\Rightarrow$ «haut niveau»
- «objet» $\not\Rightarrow$ «bas niveau»

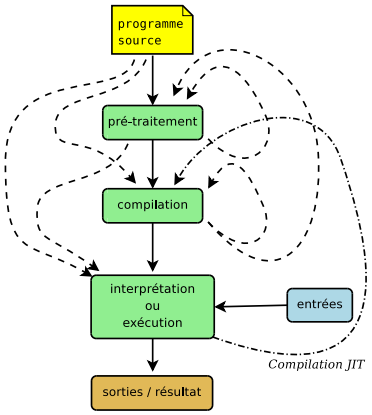
Techniques de mise en œuvre 3/4

3. Interpréter = exécuter

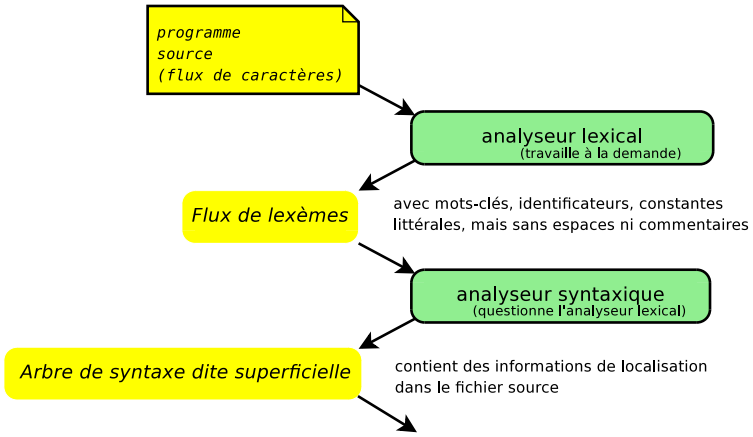
- exécuter le programme **avec** ses données
- machine logicielle (virtuelle) ou réelle



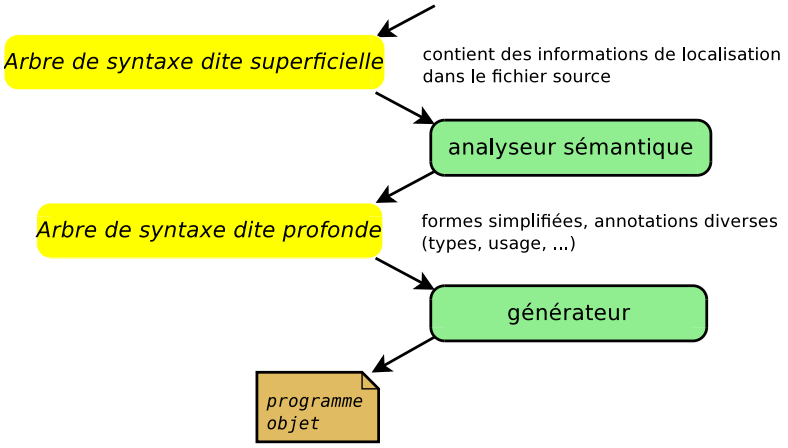
Techniques de mise en œuvre 4/4



La compilation 1/2



La compilation 2/2



OCaml

Voir le polycopié.

Le langage Objective Caml

Un peu d'histoire

1978 Robin Milner propose ML comme Méta-Langage pour décrire des stratégies dans un outil d'aide à la démonstration de théorèmes.

1981 Premières implémentations de ML.

1985 Développement de Caml à l'INRIA, de Standard ML à Edimbourg, puis à Bell Labs (New Jersey, USA).

1990 Développement de Caml-Light à l'INRIA.

1995 Ajout d'un compilateur natif et d'un système de modules.

1996 Ajout des objets.

Domaines d'utilisation

Langage algorithmique à usage général

Domaines de prédilection

- Calcul symbolique
- Prototypage rapide
- Langage de script
- Applications distribuées

Enseignement et recherche

- Classes préparatoires
- Universités (Europe, USA, Japon, . . .)

Industrie

- Startups, CEA, EDF, Dassault Aviation, Dassault Systèmes, . . .
- Intel, Microsoft, IBM, Jane Street Capital, SimCorp, Citrix (Xen), . . .

OCaml : caractéristiques

OCaml est un langage

- fonctionnel : les fonctions sont des données (presque) comme les autres
- fortement typé
- avec synthèse de types
- à gestion mémoire automatique
- compilé
- qui peut être interactif

Premiers pas

Éditer le source : bienvenue.ml

```
Printf.printf "Bienvenue !\n";;
```

Compiler et exécuter

```
$ ocamlc -o bv bienvenue.ml

$ ./bv
Bienvenue !
```


Premiers pas

En mode interactif :

```
$ ocaml
        Objective Caml version 3.11

# Printf. printf "Bienvenue !\n";;
Bienvenue !
- : unit = ()

# let euro x = floor (100.0 *. x /. 6.55957) *. 0.01;;
val euro : float -> float = <fun>

# let baguette = 5.60;;
val baguette : float = 5.6

# euro baguette;;
- : float = 0.85

# exit 0;;
```

Utilisation du mode interactif

Comme une calculette

Pour la mise au point des programmes

Comme exécutant de scripts

```
                # équivalent à la version interactive
$ ocaml < bienvenue.ml

                # idem, mais suppression des messages
$ ocaml bienvenue.ml
```


Mode Emacs

Installation : ajout de quelques lignes dans le fichier `$HOME/.emacs`

⇒ Coloration, indentation, interaction avec OCaml.

Démo ?

Programmes OCaml

Un programme est une suite de phrases de la forme

let $x = e$;;	définition de x comme résultat du calcul de e
let $f\ x_1 \ldots x_n = e$;;	définition de fonction
let rec $f\ x_1 \ldots x_n = e$;;	définition de fonction récursive
let [**rec**] $f_1\ x_1^1 \ldots x_{n_1}^1 = e_1$ **and** ... **and** $f_k\ x_n^k \ldots x_{n_k}^k = e_k$;;	définition de fonctions mutuellement récursives
e;;	expression à évaluer
type $t_1 = \ldots$ [... **and** $t_n = \ldots$]	définition de nouveaux types

où e, e_1, ..., e_k sont des **expressions**.

(∗ Commentaires (∗ imbriqués ∗) possibles ⇒ analyse lexicale des commentaires. Permet de commenter rapidement du code. ∗)

Expressions OCaml

L'expression est l'élément de base des programmes

let x = e_1 **in** e_2

let x = e_1 **and** y = e_2 **in** e_3

let x = e_1 **in let** y = e_2 **in** e_3

(* définitions locales *)

let f x = e_1 **in** e_2

let rec f x y z = ... **and** g t = ... **in** ...

let [**rec**] f [$x_1 \ldots x_n$] = e_1 [**and** ...] **in** e_2

Expressions OCaml

L'expression est l'élément de base des programmes

let [**rec**] f [$x_1 \ldots x_n$] = e_1 [**and** ...] **in** e_2 (* définition locale *)

fun $x_1 \ldots x_n$ -> e (* fonction anonyme *)

f e_1 ... e_n (* appel de fonction *)

e_0 e_1 ... e_n (* appel de fonction calculée *)

x (ou bien $M.x$ si x est défini dans M) (* identificateurs *)

(e) (* expr. entre parenthèses *)

Expressions OCaml

Structures de données : création et «déconstruction»

23, 'z', **true** (* valeurs de types de base *)

if e_1 **then** e_2 **else** e_3 (* expression conditionnelle *)

```
"INE11\n"
[| 'I', 'N', 'E', '1', '1' |]
```
(* données structurées allouées *)

e.[i] e.(i) (* accès *)

[ ] e_1::e_2 [0; 1; 3; 4] (* listes *)

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

(* analyse de cas *)

Expressions OCaml

Instructions (?)

for $i = e_1$ **to** e_2 **do** e_3 **done**
while e_1 **do** e_2 **done** (* boucles *)

e_1; e_2
(e_1; e_2) (* évaluation en séquence *)
begin e_1; e_2 **end**

Remarques

- Ni vraies instructions, ni procédures
- Boucles **for** et **while** ne fournissent pas de résultat
- Valeur () de type `unit` représente l'absence de valeur

Opérations de base

type	valeurs	opérations
`unit`	()	
`bool`	**true**, **false**	&&, ||, `not`
`char`	'a', '\n', '\065'	`Char.code`, `Char.chr`
`int`	1, 2, 3	+, −, ∗, /, `max_int`
`float`	1.0, 2., 3.14	+., −., ∗., /., `cos`
`string`	`"a\tb\010c\n"`	^, `s.[i]`, `s.[i] <- c`
tableaux	[| 0; 1; 2; 3 |]	`t.(i)`, `t.(i) <- v`
tuples	(1, 2), (1, 2, 3, 4)	`fst`, `snd`

Infixe entre parenthèses devient préfixe : (+), (∗), (**mod**)

Tableaux

Tableaux : homogènes et polymorphes

```
[| 0; 1; 3 |] : int array
[| true; false |] : bool array
```

Projections : polymorphes

```
(fun x -> x.(0)) : 'a array -> 'a
(fun t k x -> t.(k) <- x) : 'a array -> int -> 'a -> unit
```

Tableaux : toujours initialisés

```
Array.create : int -> 'a -> 'a array
Array.create 3 "chaîne" : string array
```

Tuples

Tuples : hétérogènes, mais arité fixée par le type

- une paire (1, 2) : `int * int`
- un triplet (1, 2, 3) : `int * int * int`

sont incompatibles entre eux.

Projections : polymorphes pour une arité donnée

```
(fun (x, y, z) -> x) : ('a * 'b * 'c) -> 'a
```

Paires : cas particulier avec projections prédéfinies

```
fst : ('a * 'b) -> 'a          snd : ('a * 'b) -> 'b
```


Fonctions

Arguments sont passés sans parenthèses

i

```
let plus x y = x + y        : int -> int -> int
                            : int -> (int -> int)
let plus (x, y) = x + y     : (int * int) -> int
```

La seconde fonction a un seul argument qui est une paire.

Application : simple juxtaposition

```
plus 1 3          est lu comme          (plus 1) 3
let succ = plus 1 in succ 3

Array.create (2*n) ("Hello "^"world!")
Array.create (f n) (g "abcd")
```


Fonctions récursives

Récursivité notée par le mot-clé rec

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

Récursion mutuelle

```
let rec ping n =
  if n > 0 then pong (n - 1) else "ping"
and pong n =
  if n > 0 then ping (n - 1) else "pong";;
```

Enregistrements (records, structures)

Nécessitent une déclaration

```
type monnaie = {nom : string; valeur : float};;

let x = {nom = "euro"; valeur = 6.55957};;

x.valeur;;
```

Peuvent être paramétrés

```
type 'a info = {nom : string; valeur : 'a};;

let get_val x = x.valeur;;
val get_val : 'a info -> 'a = <fun>
```


Enregistrements

Champs mutables

```
type personne = {nom : string; mutable âge : int}

let p = {nom = "Paul"; âge = 23};;
val p : personne = {nom="Paul"; âge=23}

let anniversaire x = x.âge <- x.âge + 1;;
val anniversaire : personne -> unit = <fun>

p.nom <- "Clovis";;
Characters 0-17: the label nom is not mutable
```

⇒ autorise la modification de la composante âge, mais pas de l'autre

Références

Passage d'arguments «par valeur»

- pas de &x, ∗x comme en C
- variables non modifiables
- seules valeurs modifiables : champs de structures

```
type 'a  pointeur = { mutable contenu : 'a };;
```

En C :

```
int x = 0
x = x + 1
```

En OCaml :

```
let x = {contenu = 0}
x.contenu <- x.contenu +1
```

Raccourci :

```
let x = ref 0
x := !x + 1
```

En pratique, on se sert relativement peu des références.

La ligne de commande

Arguments passés à un exécutable

- Placés dans le tableau `Sys.argv` : `string array`
- Nom de commande : `Sys.argv.(0)`
- Nombre d'arguments : `(Array.length Sys.argv) − 1`

Exemple : La commande Unix echo

`echo.ml`

```
for i = 1 to (Array.length Sys.argv) − 1 do
  Printf.printf "%s " Sys.argv.(i)
done;
Printf.printf "\n%!"; exit 0;;
```


Entrées/sorties

Canaux prédéfinis

```
stdin : in_channel
stdout : out_channel
stderr : out_channel
```

Ouverture de canaux

```
open_out : string -> out_channel
open_in : string -> in_channel
close_out : out_channel -> unit
```


Entrées/sorties

Lecture sur stdin

```
read_line : unit -> string
read_int : unit -> int
```

Écriture sur la sortie standard

```
print_string : string -> unit
print_int : int -> unit
Printf.printf : ('a, out_channel, unit) format -> 'a
```

Écriture sur la sortie d'erreurs

```
prerr_string : string -> unit
prerr_int : int -> unit
Printf.eprintf : ('a, out_channel, unit) format -> 'a
```

Entrées/sorties

Lecture sur un canal

```
input_line : in_channel -> string
input_char : in_channel -> char
```

Écriture sur un canal

```
output_string : out_channel -> string -> unit
output_char : out_channel -> char -> unit
```

Printf

```
Printf.printf
  "Bonjour %s, il %d est heures\n" nom age
Printf.eprintf ...
Printf.fprintf chan ...
Printf.sprintf ...
```


Fonctions

Fonctions : valeurs comme les autres

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

let succ n = n+1;;
let plus_deux = compose succ succ;;

let symétrique f x y = f y x;;
val symétrique : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```


Fonctions

Fonctions anonymes

```
(fun x y ->  x + y) 1 3;;
- : int = 4

let ff = ( (fun x ->  x+1), (fun x ->  x+2) );;
val ff : (int -> int) * (int -> int) = (<fun>, <fun>)
```

Notations équivalentes

```
let f(x) = e        let f x = e        let f = fun x -> e
```

Exemples

Composer n fois une fonction avec elle-même

```
let rec puissance f n =
  if n ≤ 0 then (fun x -> x) else
  let fn1 = puissance f (n − 1) in
  fun x -> f (fn1 x);;
val puissance : ('a -> 'a) -> int -> 'a -> 'a
```

Utilisations

```
let plus u v = puissance succ u v;;
let fois u v = puissance (plus u) v 0;;
let puis u v = puissance (fois u) v 1;;
puis 2 16;;
- : int = 65536
let marge = puissance (fun x -> " "^x);;
val marge : int -> string -> string
```


Unions

Unions discriminées

```
type carte = Carte of int | Valet | Dame | Roi;;
```

Constructeurs de données : commencent toujours par une majuscule.

Types paramétrés

```
type 'a réponse = Oui of 'a | Non;;
```


Unions

Types récursifs

En OCaml

```
type 'a liste =
  Vide
| Cellule of 'a * 'a liste
```

En C

```
struct cellule {
  int tête;
  struct cell *reste;
}
typedef cellule * liste;
```

Listes

Ici

```
'a liste
Vide
Cellule (e, reste)
C(e1, C(e2, C(e3, V)))
```

Listes prédéfinies

```
'a list
[ ]
e :: reste
[e1; e2; e3]
```

Unions

Définition alternative

```
type 'a liste =
  Vide
| Cellule of 'a cellule

and 'a cellule = {tête : 'a; reste : 'a liste}
```


Filtrage

Les valeurs sont souvent examinées par filtrage

```
let valeur c = match c with
  Valet -> 11
| Dame -> 12
| Roi -> 13
| Carte 1 -> 20
| Carte x -> x
```

Note : on peut aussi écrire

```
let valeur = function
  Valet -> 11
| ...
```


Filtrage

Opérer sur un type récursif de données

```
let rec longueur xs = match xs with
  Vide -> 0
| Cellule (tête, reste) -> 1 + longueur reste
```

```
let rec length xs = match xs with
  [ ] -> 0
| x :: xs -> 1 + length xs
```

Filtrage

Examen de motifs complexes

```
let f arg = match arg with
  ( [ Valet; Carte x ],     (Valet :: _ ) ) ->  ...
| ( ((Roi | Carte _) :: _), [ ] ) ->  ...
| ...
| _ ->  ...
```


Filtrage

Filtrage de champs d'enregistrements

```
let nom_de p = p.nom

let nom_de p = match p with {nom = n} -> n

let is_jean p = match p with
  {nom = ("Jean" | "jean") as n} -> true
| _ ->  false

match e with
  Oui ({valeur = Carte x} as z, _) -> z.nom
| ...
```


Filtrage : sémantique

Examen séquentiel des clauses

```
| p1 ->  e1
| ...  ->  ...
| pn ->  en
```

Filtrages incomplets, clauses redondantes

Si aucune clause ne filtre, levée d'exception.

Les motifs sont linéaires

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

Filtrages incomplets, redondants

```
let rec longueur xs =
  match xs with [ ] ->  0;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
val longueur : 'a list -> int = <fun>
```

```
let rec longueur xs = match xs with
  [ ] -> 0
| x1 :: xs -> 2
| x1 :: x2 :: xs ->  3;;
Warning U: this match case is unused.
val longueur : 'a list -> int = <fun>
```


Exceptions

Exemple

```
exception Perdu;;

let rec cherche_la_clé k xs = match xs with
  (h, v) :: t ->
     if h = k then v else cherche_la_clé k t
| [ ] -> raise Perdu;;

let k =
  try cherche_la_clé "Georges"
     [ ("Louis", 14); ("Georges", 5) ]
  with Perdu -> 10;;
```


Exceptions

Syntaxe

- Définition (phrase) — **exception** C [**of** τ]
- Levée (expression) — `raise` e
- Filtrage (expression) — **try** e **with** p_1 -> e_1 ... | p_n -> e_n

Analogie avec le filtrage de valeurs

try ... **with** ... *vs.* **match** ... **with** ...

Exceptions prédéfinies

Exception	Usage	Gravité
`Invalid_argument` **of** `string`	accès en dehors des bornes	forte
`Failure` **of** `string`	tête de liste vide renvoie le nom de la fonction	moyenne
`Not_found`	recherche infructueuse	nulle
`Match_failure` **of** ...	Échec de filtrage	fatale
`End_of_file`	Fin de fichier	nulle

Exceptions

Sémantique

- le type `exn` est une union discriminée
- levée d'exception ⇒ arrêt de l'évaluation en cours et production d'une valeur «exceptionnelle» (du type `exn`)
- traitement d'exception possible seul[t] si levée durant l'évaluation de e dans **try e with** p_1 `->` e_1 `|` ... `|` p_n `->` e_n :
  - si e produit v, on renvoie v
  - si e lève xc, filtrage de xc par p_1 `->` e_1 `|` ... `|` p_n `->` e_n
    - si p_i est le premier à filtrer xc, on renvoie la valeur de e_i
    - sinon, on propage (*raise*) xc à la recherche d'un **try** ... **with** ... englobant

Exceptions

On peut observer une exception (l'attraper et la relancer)

```
try e with
  (Failure s) as x ->
    begin
      close_in inchan;
      close_out outchan;
      Printf.eprintf "Warning: %s\n" s;
      raise x
    end
```


Exceptions

Usage

Définir de nouvelles exceptions plutôt que réutiliser les exceptions prédéfinies

On utilise les exceptions pour

- reporter une erreur et interrompre le calcul en cours ; par exemple `Match_failure`
- communiquer une valeur de retour rare ou particulière ; exemple : `Not_found` et `End_of_file`

Exemple

La commande Unix `cat` (version restreinte)

```
try
  while true do
    print_string (read_line());
    print_newline()
  done
with End_of_file -> ();;
```


Éléments de style I : cascades

Cascades de `let`

```
let x = ... in
faire qquechose;
let y = ... x ... in
let f z t u = ... (x,y,z) in
...
```

Cascades de `let`

```
let x = ... in
let _ = faire qquechose in
let y = ... x ... in
let f z t u = ... (x,y,z) in
...
```

Cascades de `if`

```
if ... then ... else
if ... then (...; ...) else
if ... then begin ... end
else ...
```

Cascades de `match/try`

```
match ... with
    ... -> -> f(x); g(y)
| ... -> (match ... with ...)
| ... ->
  begin match ... with ... end
```


Éléments de style II : exception *vs.* option

Représenter succès/échec : le type `option`

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

Exemple

```
let rec cherche k xs = match xs with
  (clé, v) :: ys ->
    if clé = k then Some v else cherche k ys
| [ ] -> None;;
val cherche : 'a -> ('a * 'b) list -> 'b option = <fun>
```

```
match cherche "Georges" participants with
  None ->  ...
| Some v ->  ... v ...
```

Forme souvent une alternative viable aux exceptions

À rapprocher de `NULL` en C

Aides à la mise au point

Tracer une fonction : seulement en mode interactif

#*commande* [*arguments*] ;;

Mise en œuvre

```
#trace fact ;;
fact is now traced.
fact ⟵ 5
fact ⟵ 4
fact ⟵ 3
fact ⟵ 2
```

```
fact ⟵ 1
fact ⟶ 1
fact ⟶ 2
fact ⟶ 6
fact ⟶ 24
fact ⟶ 120
- : int = 120

#untrace fact ;;
#untrace_all ;;
```

Et :

- débogueur
- fonctions d'impression de valeurs
- ... et bien sûr `Printf.printf`

Polymorphisme

Souvent plusieurs types possibles pour une même valeur :

```
let id x = x;;

id : int -> int
id : string -> string
id : ('a * 'a) -> ('a * 'a)
id : ('a -> 'b) -> ('a -> 'b)
id : 'a -> 'a
```

Type plus ou moins général

- il existe un «meilleur» type pour `id`, dont tous les autres sont **instances**
- ∀ `'a` . `'a -> 'a`
- instances : `int -> int`, `('b -> 'b) -> ('b -> 'b)`

Polymorphisme

Le `let` est l'unique occasion de produire du polymorphisme

- **let** `id x = x` **in** (`id` **true**, `id 3`) — oui
- (**fun** `id ->` (`id` **true**, `id 3`)) (**fun** `x ->` `x`) — **non**

Le `let` sans calcul est l'occasion de produire du polymorphisme

```
let id =
  let z = calcul() in
  (fun x -> x) in
(id true, id 3)
```

— **non**

Polymorphisme

Types non complètement déterminés ('_a, '_b,...)

- Dans la boucle interactive :

```
let id x = x;;
val id : 'a -> 'a = <fun>
let f = id id;;
val f : '_a -> '_a = <fun>
```

- `f` **n'est pas polymorphe** : son type n'est pas complètement déterminé

```
f (1,2);;
- : int * int = (1, 2)
f;;
- : int * int -> int * int = <fun>
```


Modules

Programmation modulaire

- découpage du programme en morceaux compilables indépendamment
  pour rendre les gros programmes compilables
- donner de la structure au programme
  pour rendre les gros programmes compréhensibles
- spécifier les liens (interfaces) entre les composantes
  pour rendre les gros programmes maintenables
- identifier des sous-composantes indépendantes
  pour rendre les composantes réutilisables

Modules de base

Structures : collections de phrases

```
struct
  p1
  ...
  pn
end
```

```
struct
  let x = ... ;;
  module M = struct ... end;;
  module type S = sig ... end;;
  type ('a, 'b) t = ...
  exception Ok;;
end
```

Les p_i sont des

- phrases du langage (déclarations de types, de valeurs, d'exceptions)
- définitions de sous-modules **module** X = ...
- définitions de types de module **module type** S = ...

Modules

Création de module

```
module S = struct
  type t = int
  let x = 0
  let f x = x+1
end
```

```
module T = struct
  type t = int
  let x = 0
  module U = struct
    let y = true
  end
  let f x = x+1
end
```


Modules

Utilisation d'un module

référence aux composantes d'un module par *M.comp*

```
... (S.f S.x : S.t) ...
```

Directive open

- alternative à la notation pointée
- pour utilisation intensive d'un module particulier

```
open S

... (f x : t) ...
```


Modules

Un module peut être composante d'un autre module

```
module T = struct
  module R = struct
    let x = 0
  end
  let y = R.x + 1
end
```

La notation pointée s'étend en *chemin* :

```
module Q = struct
  let z = T.R.x
  open T.R        donne accès aux composantes de T.R
    ...
end
```

Modules

Signatures : types de modules = collections de «spécifications»

Signature

sig
  $spec_1$
  ...
  $spec_n$
end

où chaque $spec_i$ **est de la forme :**

val `x : s`	valeurs
type `t`	types abstraits
type `t` $= \tau$	types manifestes
exception `E`	exceptions
module `X : M`	sous-modules
module type `S` [= `M`]	types de module

Nommage d'un type de module par **module type** `T` = ...

```
module type MYSIG = sig type t val x : t end
```


Modules

Le compilateur infère les signatures de modules

Module

```
module Exple = struct
  type t = int
  module R = struct
    let x = 0
  end
  let y = R.x + 1
end;;
```

Signature inférée

```
module Exple : sig
  type t = int
  module R : sig
    val x : int
  end
  val y : int
end
```


Modules

Masquage par signature : (*structure* : *signature*)

- vérifie que la structure **satisfait** la signature
- rend inaccessibles les parties de la structure qui ne sont pas spécifiées dans la signature ;
- produit un résultat qui peut être lié par **module** `M` = ...

Sucre syntaxique :

module `M` : *signature* = *structure*

équivaut à

module `M` = (*structure* : *signature*)

Modules

Restriction équivalentes

```
module S = (struct
  type t = int
  let x = 1
  let y = x + 1
end : sig
  type t
  val y : t
end);;
```

```
module type SIG = sig
  type t
  val y : t
end

module S : SIG = struct
  type t = int
  let x = 1
  let y = x + 1
end;;
```

Utilisation

```
S.x;;                Error: S.x is unbound
S.y + 1;;            Error: S.y is not of type int
```


Modules et compilation séparée

Une unité de compilation A se compose de deux fichiers :

1. implémentation `A.ml` :
   - une suite de phrases
   - semblable à un intérieur de **struct** ... **end**
2. interface `A.mli` (optionnel) :
   - une suite de spécifications
   - semblable à l'intérieur de **sig** ... **end**

Une autre unité de compilation `B` peut faire référence à `A` comme si c'était une structure, en utilisant la notation pointée `A.x` ou bien en faisant **open** `A`

Compilation séparée

Fichiers sources : `a.ml`, `a.mli`, `b.ml`

Étapes de compilation :

```
ocamlc -c a.mli   compile l'interface de A
                  crée a.cmi
ocamlc -c a.ml    compile l'implémentation de A
                  crée a.cmo
ocamlc -c b.ml    compile l'implémentation de B
                  crée b.cmo
ocamlc -o monprog a.cmo b.cmo
                  édition de liens finale
                  crée monprog
```

Modules paramétrés

Foncteur : fonction des modules dans les modules

Corps du foncteur explicitement paramétré par le paramètre de module S.

functor (*S* : *signature*) -> *module*

Exemple

```
module T = functor(S : SIG) -> struct
  type u = S.t * S.t
  let  y = S.g(S.x)
end
```


Application de foncteur

```
module T1 = T(S1)
module T2 = T(S2)
```

T1 et T2 s'utilisent alors comme des structures ordinaires :

```
(T1.y : T2.u)
```

T1 et T2 partagent entièrement leur code.