Initiation au langage OCaml.

Didier Rémy
Janvier 2006

http://pauillac.inria.fr/~remy/ocaml/./
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/ocaml/./

Travaux Dirigés avec Fabrice Le Fessant et Maxence Guesdon
avec les contributions significatives de
Luc Maranget, David Monniaux, et Benjamin Monate

Slide 1

Calendrier

Jour	Heure	Groupe 1	Groupe 2
Mardi 03	08 :30 - 09 :00	présentation Majeure	
	09 :15 - 10 :45	Cours Caml	
	11 :00 - 12 :30	Cours C	
Mardi 03	13 :45 - 15 :45	Caml	C
	16 :00 - 18 :00	C	Caml
Mercredi 04	08 :00 - 10 :00	Caml	C
	10h :15 - 12 :15	C	Caml
Vendredi 06	13 :45 - 15 :45	C	Caml
	16 :00 - 18 :00	Caml	C
Mercredi 11	08 :00 - 10 :00	Caml	C
	10h :15 - 12 :15	C	Caml

Motivation

Pour l'enseignement

Le langage OCaml est utilisé en particulier
– Langages de Programmation (majeure 1)
– Modularité, Objets, et Types (majeure 1)
– Programmation Système (majeure 2)

Slide 2

– Compilation (majeure 2)
+ Certains stages d'options en informatique.

Après l'X

– Connaître différents langages de programmation pour choisir le mieux approprié dans chaque situation.
– Le langage OCaml (où d'autres langages de la même famille) sont de plus en plus utilisés dans l'industrie.
  Voir les infos en ligne, *e.g.* FFT. Récemment choisi pour des outils système de la distibution Lindow de Linux !

Avertissement

Cette introduction succincte au langage OCaml a pour but de vous permettre de vous débrouiller avec le langage OCaml.

Ce n'est qu'un survol du langage, vu sous l'angle de la pratique.
– Les exemples sont simples et classiques.
– Les constructions de bases sont présentées, mais avec très peu d'exemples.

Slide 3

– Les constructions particulières au langage OCaml telles que l'utilisation des fonctions d'ordre supérieur, du filtrage ou du polymorphisme mériteraient plus de considération et d'exercices.

Il doit vous permettre de vous exercer ensuite :
– en suivant les travaux dirigés,
– dans les autres cours (langages de programmation, compilation, programmation système), ou
– seul, à partir des nombreux exercices en ligne.

Apprentissage d'un langage

Slide 4

Par la pratique exercez-vous !

Par couper/coller
- regarder les solutions des exercises ;
- réutiliser en le modifiant pour l'adapter du code déjà écrit,
- y compris le vôtre.

En se référant à la définition du langage
- la documentation : le manuel de référence, la librairie du noyau et les librairies standards ;
- le tutorial, l'expérience des autres (FAQ) ;
- introspection d'OCaml ;
- les librairies développées par les utilisateurs (voir aussi la page ocaml).

Profiter des informations en ligne rassemblées dans la bosse.

La petite histoire

Slide 5

OCaml est le fruit de développements récents et continus :

1975 Robin Milner propose ML comme méta-langage (langage de script) pour l'assistant de preuve LCF.
Il devient rapidement un langage de programmation à part entière.

1981 Premières implantations de ML

1985 Développement de Caml à l'INRIA. et en parrallèle, de Standard ML à Edinburgh, de "SML of New-Jersey", de Lazy ML à Chalmers, de Haskell à Glasgow, etc.

1990 Implamtation de Caml-Light par X. Leroy and D. Doligez

1995 Compilateur vers du code natif + système de modules

1996 Objets et classes (OCaml)

2001 Arguments étiquettés et optionnels, variantes.

2002 Méthodes polymorphes, librairies partagées, *etc.*

2003 Modules récursifs, private types, *etc.*

Domaines d'utilisation du langage OCaml

Slide 6

Un langage d'usage général avec des domaines de prédilection.

Domaines de prédilection

– Calcul symbolique : Preuves mathématiques, compilation, interprétation, analyses de programmes.
– Prototypage rapide. Langage de script. Langages dédiés.
– Programmation distribuée (bytecode rapide).

Enseignement et recherche

– Classes préparatoires.
– Utilisé dans de grandes universités (Europe, US, Japon, etc.).

Domaines d'utilisation du langage OCaml

Slide 7

Mais aussi…

Industrie

– Startups, CEA, EDF, France Telecom, Simulog,...

Gros Logiciels

– Coq, Ensemble, ASTREE, (voir la bosse d'OCaml)

Plus récemment, outils système

– Unison,
– MLdonkey (peer-to-peer),
– Libre cours (Site WEB),
– Lindows (outils système pour une distribution de Linux)

Quelques mots clés

Slide 8

Le langage OCaml est :

- *fonctionnel* : les fonctions sont des valeurs de première classe, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions.
- *à gestion mémoire automatique* (comme JAVA)
- *fortement typé* : le typage garantit l'absence d'erreur (de la machine) à l'exécution.
- *avec synthèse de types* : les types sont facultatifs et synthétisés par le système.
- *compilé ou interactif* (mode interactif = grosse calculette)

De plus le langage possède :

- *une couche à objets* assez sophistiquée,
- *un système de modules* très expressif.

Premiers pas en OCaml.

Le mode compilé

Éditer le source

hello.ml

```
print_string "Hello World!\n";;
```

hello.ml

Slide 9

Compiler, lier et exécuter

Sous le Shell d'Unix

```
ocamlc -o hello hello.ml
./hello
```

Hello World!

Sous le Shell d'Unix

Le mode interprété

Slide 10

Mode interactif avec l'utilisateur :

Sous le shell d'Unix

```
ocaml
```

Objective Caml version 3.06

```
print_string "Hello World!\n";;
```

Hello World!
- : unit = ()

```
let euro x = floor (100.0 *. x /. 6.55957) *. 0.01;;
```

val euro : float -> float = <fun>

```
let baguette = 4.20;;
```

val baguette : float = 4.2

```
euro baguette;;
```

- : float = 0.64

```
exit 0;;
```

Sous le shell d'Unix

Utilisation du mode interprété

Comme une grosse calculette

Pour la mise au point des programmes

Évaluation des phrases du programme en cours de construction (utilisation avec un éditeur)

Slide 11

Comme langage de script...

`% ocaml < bienvenue.ml`	équivalent à la version interactive
`% ocaml bienvenue.ml`	idem, mais suppression des messages

Conseil *Il faut savoir que le mode interprété existe, qu'il peut parfois aider à la mise au point, mais à part pour ces usages très particuliers, nous le déconseillons de façon générale, surtout pour les débutants qui peuvent être déroutés par l'évaluation incrémentale des phrases d'un programme.*

On peut également en faire un script directement exécutable en indiquant le chemin absolu de `ocaml` *précédé de* `#!` *sur la première ligne du fichier :*

bonjour

```
#!/usr/bin/ocamlrun /usr/bin/ocaml

print_string "Bonjour!"; print_newline();;
```

bonjour

Sous le Shell d'Unix

```
chmod +x bonjour
bonjour
```

Sous le Shell d'Unix

Slide 12

Le mode caml pour Emacs

Slide 13

Installation (si nécessaire)

Insérer le contenu de `~remy/emacs/ocaml.emacs`
à la fin de votre fichier `~/.emacs`
(Quitter `emacs` après l'installation.)

Édition et compilation

- le mode OCaml doit être sélectionné automatiquement à l'ouverture d'un fichier se terminant avec le suffixe `.ml` ou `.mli`. Par exemple, ouvrir un fichier `bienvenue.ml` et un menu Caml doit apparaître dans la bar de menus.
  Écrire les phrases dans ce fichier.
- Exécuter la comande Compile dans le menu Caml (ou son équivalent clavier CONTROL-C CONTROL-C). Modifier si besoin la commande proposée par défaut.
  Tapez RETURN.

Slide 14

- Documentation interactive : placer le curseur sur ou juste après un identificateur. Appeler la commande Help for identifier dans le menu Caml (ou son équivalent clavier META-CONTROL-H).

Version interactive : au choix

- Ouvrir un fichier `bienvenue.ml`,
  Écrire les phrases dans ce fichier,
  Exécuter la commande Eval phrase dans le menu Caml.
  (éventuellement, appeler Start subshell dans le menu Caml.)
- Appeler directement OCaml par la commande Run caml

Mode interprété

Mode vi déconseillé : les accrocs sont laissés à eux-mêmes.

Slide 15

Les programmes

Un *programme* est une suite de *phrases* :

- définition de valeur — `let` $x = e$
- définition de fonction — `let` $f\ x_1\ ...\ x_n = e$
- définition de fonctions mutuellement récursives — `let` [rec] $f_1\ x_1\ ... = e_1\ ...$ [`and` $f_n\ x_n\ ... = e_n$]
- définition de type(s) — `type` $q_1 = t_1 ...$ [`and` $q_n = t_n$]
- expression — e

Les phrases se terminent par `;;` optionnel entre des déclarations, mais obligatoires sinon.

```
(* Cela est un commentaire (* et ceci un commentaire à
l'intérieur d'un commentaire *) sur plusieurs lignes *)
```

Slide 16

Les expressions

- définition locale — `let` $x = e_1$ `in` e_2
  (définition locale de fonctions mutuellement récursives)
- fonction anonyme — `fun` $x_1\ ...\ x_n$ `->` e
- appel de fonction — f $e_1\ ...\ e_n$
- variable — x ($M.x$ si x est défini dans le module M)
- valeur construite — (e_1, e_2)
  dont les constantes — 1, 'c', "aa"
- analyse par cas — `match` e `with` p_1`->`$e_1 \ldots \mid p_n$`->`e_n
- boucle for — `for` $i = e_0$ `to` e_f `do` e `done`
- boucle while — `while` e_0 do e `done`
- conditionnelle — `if` e_1 `then` e_2 `else` e_3
- une séquence — $e; e'$
- parenthèses — (e) `begin` e `end`

Un vrai petit programme

Slide 17

ls.ml

```
let all =  Sys.readdir ".";;
let filter p =
  Array.fold_left
    (fun res x -> if p x then x :: res else res) [];;
let has_suffix suf str =
  let len_suf = String.length suf in
  let len_str = String.length str in
  len_suf <=  len_str &&
  String.sub str (len_str - len_suf) len_suf = suf;;

List.iter print_endline (filter (has_suffix ".ml") all);;
```

ls.ml

Exercise 1 *Que fait ce code ?* Answer

Le modifier le code pour filtrer selon un préfixe donné. □

Remarques

Slide 18

Il n'y a ni instructions ni procédures.
Toutes les expressions retournent une valeur.

– La valeur unité `()` de type `unit` ne porte aucune information (c'est l'unique valeur possible dans son type).

  `print_int : int -> unit`

  `print_string : string -> unit`

– L'expression e dans les boucles `for` et `while` et dans la séquence doit être de type `unit`.

– Si nécesaire, jeter explicitement le résultat, en utilisant

  – une fonction (prédéfinie) :

  ```
  let ignore x = ();;
  ```

  ignore : 'a –> unit

  ```
  ignore e1; e2 : unit
  ```

  – ou une définition :

  ```
  let _ =  e1 in e2
  ```

Slide 19

Ne pas confondre

La liaison toplevel

```
let x = 1;;
let f x = x + 1;;
let y = f x;;
```

les ;; intermédiaires sont optionnels

La liaison locale

```
let y =
  let x = 1 in
  let f x = x + 1 in
  f x;;
```

Une seule phrase toplevel et deux liaisons locales

La séquence

```
let sequence =
  print_string "vitesse = ";
  print_int 10;
  print_newline ()
```

Dans une séqeunce les ; sont obligatoires. Les expressions de la séquences ne sont pas liées. La valeur de la dernière expression est retournée.

Slide 20

Types, constantes et primitives de base

`unit`	`()`	`pas d'opération!`
`bool`	`true` `false`	`&&` `||` `not`
`char`	`'a'` `'\n'` `'\097'`	`code` `chr`
`int`	`1` `2` `3`	`+` `-` `*` `/` `max_int`
`float`	`1.0` `2.` `3.14`	`+.` `-.` `*.` `/.` `cos`
`string`	`"a\tb\010c\n"`	`^` `s.[i]` `s.[i] <- c`

Polymorphes

`tableaux`	`[| 0; 1; 2; 3 |]`	`t.(i)` `t.(i) <- v`
`tuples`	`(1, 2)` `(1, 2, 3, 4)`	`fst` `snd`

Un opérateur infixe entre parenthèses devient préfixe. Par exemple, `(+)` x_1 x_2 est équivalent à x_1 `+` x_2

Tableaux

Les opérations sur les tableaux sont polymorphes : on peut créer des tableaux *homogènes* d'entiers, de booléens, etc.

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

Les indices de tableaux varient de 0 à $n-1$ où n est la taille.

Slide 21

Les projections sont *polymorphes* : elles opèrent aussi bien sur des tableaux d'entiers que sur des tableaux de booléens.

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

Les tableaux sont toujours initialisées :

`Array.create`	`: int -> 'a -> 'a array`

- Le premier argument est la longueur du tableau

- Le second argument est utilisé pour initialiser toutes les cases du tableau. Il détermine le type du tableau.

Slide 22

Tableaux (examples)

```
let digits = Array.create 10 0;;
```

```
val digits : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
let _ =
  for i = 0 to 9 do digits.(i) <- digits.(i) + i done;;
let chriffre_trois = digits.(3);;
```

```
val chriffre_trois : char = '3'
```

Slide 23

Polymorphisme :

```
Array.set;;
```

```
- : 'a array -> int -> 'a -> unit = <fun>
```

```
Array.set digits;;
```

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

Tuples

Ils sont hétérogènes, mais leur arité est fixée par leur type :

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

sont incompatibles.

Slide 24

Les projections sont polymorphes sur les tuples de même arité :

```
let second_of_triple (x, y, z) = y
```

```
second_of_triple : 'a * 'b * 'c -> 'b
```

Les paires ne sont qu'un cas particulier de tuples, pour lequel les deux projections `fst` et `snd` sont pré-définies.

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

Les autres projections doivent être programmées (*c.f.* ci-dessus).

Les fonctions

Les arguments sont passés *sans parenthèses*

Slide 25

Écriture recommandée

```
let middle x y = (x+y) / 2
```

val middle :
int -> int -> int = <fun>

```
milieu 5 11;;
```

Mauvaise écriture (*)

```
let middle (x,y) = (x+y) /2
```

val middle :
*int * int -> int = <fun>*

(*) À droite "`milieu`" prend un seul argument qui est une paire.

Fonctions récursives

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

... et mutuellement récursives

```
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";;
```

Applications

Exercise 2 *Quelle différence y a-t-il entre (lorsque* ***f****,* ***x****,* ***y*** *sont des variables) :*

f x y ***(f x) y*** ***f (x y)*** ***(f(x))(y)***

Answer ☐

Slide 26

Application partielle

Une fonction à deux arguments peut n'en recevoir qu'un seul.

```
let plus x y = x + y;;
```

val plus : int -> int -> int = <fun>

```
let incr2 = plus 2;;
```

val incr2 : int -> int = <fun>

La fonction `incr2` est équivalente à `fun y -> plus 2 y` Remarque : on aurait pu écrire :

```
let plus = ( + );;
```

Abstraction

Exercise 3 *On suppose que* f *est de type* `int -> int -> int`. *Quelle différence y a-t-il entre :*

```
f          fun x -> f x          fun x y -> f x y
```

Montrer le cas échéant la différence sur un exemple.

Slide 27

Indication : on pourra choisir une fonction `f` qui fait des effets de bord

Answer □

Les enregistrements

Il faut les déclarer au préalable

```
type monnaie = {nom : string; valeur : float}
let x = {nom = "euro"; valeur =  6.55957}
x.valeur
```

Analogue à des tuples à champs nommés.

Slide 28

Champs polymorphes

```
type 'a info = {nom : string; valeur : 'a};;
fun x -> x.valeur;;
```

```
- : 'a info -> 'a
```

Attention ! un nom peut en cacher un autre !

L'étiquette `nom` désigne la projection de l'enregistrement `'a info`, le champ `nom` du type `monnaie` dernier est devenu inaccessible.

Les structures mutables

Slide 29

Seuls les champs déclarés mutable au moment de la définition pourront e6tr emodifier.

Par exemple

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

permettra de modifier le champ `age`, mais pas le champ `nom`.

```
let p = {nom = "Paul"; age = 23};;
```

```
val p : personne = {nom = "Paul"; age = 23}
```

```
let anniversaire x = x.age <- x.age + 1;;
```

```
val anniversaire : personne -> unit = <fun>
```

```
p.nom <- "Clovis";;
```

```
Characters 0-17:
The label nom is not mutable
```

Les références

Slide 30

Le passage d'arguments se fait toujours par valeur.

Les constructions `&x` ou `*x` de C qui retourne l'adresse à laquelle se trouve une valeur ou la valeur qui se trouve à une adresse n'existent pas en OCaml.

La valeur d'une variable n'est pas modifiable

Mais, on peut modifier les champs d'un enregistrement. Le système a préféfini le type enregistrement suivant :

```
type 'a ref = { mutable contents : 'a }
```

En C/Java	En OCaml :	Équivalent à :
`int x = 0;`	`let x = ref 0;;`	`let x = {contents = 0};;`
`x = x + 1;`	`x := !x +1;;`	`x.contents <- x.contents +1;;`

Peu de valeurs ont en fait besoin d'être des références en OCaml. Éviter les références inutiles. Le style «fonctionnel» est plus sûr.

Slide 31

Égalité

Égalité structurelle (notée `=` et `<>` pour l'inégalité)

C'est l'égalité mathématique : deux valeurs sont égales si elles ont la même structure et si leurs sous-structures respectives sont égales.

```
1 = 1  && [|2|] = [|2|] && "un" = "un" && ref 1 = ref 1
```

− : bool = true

Slide 32

Égalité physique (notée `==` et `!=` pour l'inégalité)

C'est l'égalité de l'emplacement mémoire. Dépend de la représentation des valeurs. Les valeurs mutables sont toujours allouées. L'égalité physique implique l'égalité structurelle.

```
[] = [] && [ 1 ]  != [ 1 ] && "eq" != "eq" && 1.0 != 1.0
&& ref 1 != ref 1 && (let x = ref 1 in x == fst (x, x))
```

− : bool = true

Égalité (exercice)

Exercise 4 *Quelles sont les égalitées vraies ?*

```
1 == 1;;
1L == 1L;;
nan = nan;;
nan == nan;;
'a' == 'a';;
type nat = Zero | Succ of nat;;
Zero == Zero;;
Succ Zero == Succ Zero;;
```

Slide 33

Answer ☐

- Utilisez l'égalité structurelle de préférence.
- Réservez l'égalité physique aux cas où c'est strictement ce que vous voulez.
- Mieux, définissez votre propre égalité !

Les arguments de la ligne de commande

Les arguments passés à un exécutable

sont placés dans le tableau `Sys.argv : string array`

Le nom de la commande est compté (c'est le premier argument).

Exemple La commande Unix `echo`

Slide 34

echo.ml

```
for i = 1 to Array.length Sys.argv - 1
do print_string Sys.argv.(i); print_char ' ' done;
print_newline();;
```

echo.ml

Exercise 5 (Commade echo) *Corriger le programme ci-dessus pour qu'il reconnaisse, comme la commande Unix `/bin/echo`, l'option `"-n"` si elle est passée en premier argument, ce qui a pour effet de ne pas imprimer de fin de ligne. Corriger également l'impression de l'espace final (qui ne doit pas être imprimé).* Answer ☐

Usage avancé

La librairie `Arg` permet de lire ces arguments plus facilement.

Slide 35

Retourner un résultat

La fonction `exit` : `int -> unit`

Elle arrête le programme et retourne au système d'exploitation la valeur entière reçue en argument (modulo 256).

Par défaut, une exécution retourne la valeur 0 si elle se termine normalement et une valeur non nulle autrement (lorsqu'une exception s'échappe, par exemple)

Slide 36

L'usage est de retourner 0 pour une évaluation normale, et de réserver les valeurs non nulles pour décrire un comportement anormal. (Sous unix on peut observer le code de retour d'une commande en exécutant `echo $?` immédiatement après.)

Exercise 6 (Commandes true et false) *Écrire deux programmes* ***`true`*** *et* ***`false`*** *qui se comporte comme les commandes Unix* ***`/bin/true`*** *et* ***`/bin/false`***, i.e. *qui ne font rien et retourne avec les codes 0 et 1, respectivement).* Answer □

Les entrées-sorties

Slide 37

Canaux pré-définis

`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`

Lecture sur stdin

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

Écriture sur stdout

`print_string`	: `string -> unit`
`print_int`	: `int -> unit`

Les entrées sorties avancées

– Voir la librairie noyau

– Voir la librairie Printf par exemple, on écrira comme en C :

```
Printf.printf "%d %s %d = %d\n" 2 "+" 3 (2+3);;
```

La puissance des fonctions

Slide 38

Les fonctions sont des valeurs comme les autres

– En particulier, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions.

– Les fonctions se manipulent comme en mathématiques.

La composition de deux fonctions est une fonction

```
let compose f g = fun x -> f (g x);;
```

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

Le type de fonction compose peut se reconstituer ainsi : Le premier argument `f` est une fonction quelconque, donc de type `('a -> 'b)`. Le deuxième argument `g` est une fonction dont le résultat doit pouvoir être passé à `f` donc de type `'a` ; le domaine de `g` est quelconque ; donc `g` est de type `'c -> 'a`. Le résultat est une fonction qui prend un argument `x` qui doit pouvoir être passé à `g` donc du type `'c`. Finalement, le résultat final sera retourné par `f`, donc de type `'b`.

Fonctions anonymes

Ci-dessus, la fonction `fun x -> f (g x)` est anonyme.

```
let compose = fun f -> fun g -> fun x -> f (g x)
let compose f g x = f (g x);;
```

sont des définitions équivalentes de la fonction `compose`.

Slide 39

La fonction puissance

Fonction puissance

```
let rec power f n =
  if n <= 0 then (fun x -> x)
  else compose f (power f (n-1));;
```

```
val power : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Slide 40

Dérivation

```
let derivative dx f = fun x -> (f(x +. dx) -. f(x)) /. dx;;
```

```
val derivative :
  float -> (float -> float) -> float -> float = <fun>
```

La dérivée $n^{\text{ème}}$ est la puissance n de la dérivée. Soit !

```
let pi = 4.0 *. atan 1.0;;
let sin''' = (power (derivative 1e-5) 3) sin in sin''' pi;;
```

```
- : float = 0.99999897...
```

à la précision des calculs numériques près...

Les variantes et le filtrage

Les définitions de types sommes

Analogues aux définitions de types d'enregistrements.

Par exemple, les booléens sont éléments du type suivant :

```
type boolean = Vrai | Faux;;
let t = Vrai and f = Faux;;
```

Slide 41

Les constructeurs sont toujours capitalisés

Le filtrage permet d'examiner les valeurs d'un type somme.

```
let int_of_boolean x =
  match x with Vrai -> 0 | Faux -> 1;;
```

Raccourci (notation équivalente)

```
let int_of_boolean = function Vrai -> 0 | Faux -> 1;;
```

Le type option

Les références sont toujours initialisées en OCaml. Il peut être nécessaire de créer des références qui seront définies plus tard.

Ce traitement doit être explicite.

Prédéfini par le système

```
type 'a option = Some of 'a | None
let statut = ref None;;
```

Slide 42

Plus tard, on pourra définir la valeur :

```
statut := Some 3;;
```

La lecture du statut doit aussi considéré le cas où celui-ci est indéfini.

```
match !statut with Some x -> x | None -> 0;;
```

Les listes

Les (fausses) listes sont simplement un type somme récursif

Types récursifs

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

Fausses	Vraies
'a liste	'a list
Vide	[ ]
Cellule (t,r)	t :: r

Slide 43

Définition alternative (utilisant un enregistrement)

```
type 'a liste = Vide | Cellule of 'a cell
and 'a cell = { t^^eate : 'a; reste : 'a liste };;
```

Listes modifiables en place (les champs sont mutables)

```
type 'a liste = Vide | Cellule of 'a cell
and 'a cell = { mutable hd : 'a; mutable tl : 'a liste };;
```

Les vraies listes (librairie List)

Ce sont les listes qu'ils faut utiliser ! Le système pré-définit :

```
type 'a list = [] | (::) of 'a * 'a list
```

Racourci Les trois formes suivantes sont équivalentes :

```
let l = 1 :: (2 :: (3 :: (4 :: [])));;
let l = 1 :: 2 :: 3 :: 4 :: [];;
let l = [1; 2; 3; 4];;
```

Slide 44

La fonction reverse

Renforcer l'hypothèse de récurrence : Une fonction auxilliaire transfert les éléments d'une liste dans l'autre.

Slide 45

```
let rec transfert dest source =
  match source with
    [] -> dest
  | head :: rest -> transfert (head :: dest) rest;;
let reverse l = transfert [] l;;
```

Écriture équivalente (avec une fonction locale)

```
let reverse l =
  let rec transfert dest source =
    match source with
      [] -> dest
    | head :: rest -> transfert (head :: dest) rest
  in transfert [] l;;
```

Exercise 7 (Listes à double entrée) *On donne le type* `double` *des listes à "double entrée", qui possède deux version* `Gauche` *et* `Droite` *du constructeur* `Cellule`*.*

```
type 'a double =
    Vide
  | Gauche of 'a * 'a double
  | Droite of 'a double * 'a;;
```

On pourra ainsi écrire

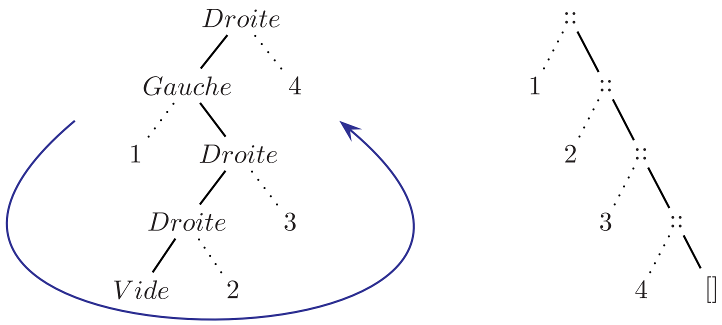
```
let d = Droite (Gauche (1, Droite (Droite (Vide, 2), 3)), 4);;
```

Slide 46

La représentation d'une liste à double entrée est une arbre filiforme (figure de Gauche). On peut lire une liste à double entrée en rebalançant l'arbre à doite (figure de droite), ce qui revient à faire un parcours gauche droite des feuilles (dans la figure de droite, on a remplacé les noeuds

`Gauche` et `Vide` par ceux des “vrais” listes `::` et `[]`).

Slide 47



Écire une fonction `print_double` imprime les noeuds d'une liste double dans un parcours gauche-droite (`print_double` reçoit en premier argument une fonction d'impression d'un nœud). Answer

En remplaçant la fonction d'impression au terminal par une fonction qui stocke les éléments dans une (vraie) liste, en déduire une fonction `list_of_double` qui transforme une liste double en une (vraie) liste simple (comme illustré par le passage de la figure de gauche à la figure de droite). Answer

Écrire une autre version de la fonction `d^^e9double` qui n'utilise pas de référence (on s'inpirera de la fonction `reverse` ci-dessus). Answer

Utiliser la trace pour observer l'exécution de la fonction `d^^e9double`. (On aura intérêt à tracer une version spécialisée pour les entiers) :

Slide 48

```
let d^^e9double_entiers =
   (d^^e9double : int list -> int double -> int list)
let list_of_double_entiers l = d^^e9double_entiers [] l;;
#trace d^^e9double_entiers;;
list_of_double_entiers d;;
```

□

Le jeu de carte

Il combine types sommes et types enregistrements :

```
type carte = Carte of ordinaire | Joker
and ordinaire = { couleur : couleur; figure : figure; }
and couleur = Coeur | Carreau | Pic | Trefle
and figure =  As | Roi | Reine | Valet | Simple of int;;
```

Définition de fonctions de construction auxiliaires :

Slide 49

```
let valet_de_pic = Carte { figure=Valet; couleur=Pic; }
```

val valet_de_pic : carte = Carte {couleur=Pic; figure=Valet}

```
let carte n s = Carte {figure = n; couleur = s}
let roi s = carte Roi s;;
```

val carte : figure −> couleur −> carte = <fun>
val roi : couleur −> carte = <fun>

Filtrage en profondeur

```
let valeur x =
  match x with
  | Carte { figure = As } -> 14
  | Carte { figure = Roi } -> 13
  | Carte { figure = Reine } -> 12
  | Carte { figure = Valet } -> 11
  | Carte { figure = Simple k } -> k
  | Joker -> 0;;
```

Slide 50

Un motif peut capturer plusieurs cas.

```
let petite x =
  match x with
  | Carte { figure = Simple _ } -> true
  | _ -> false;;
```

Slide 51

Sémantique du filtrage

Lorsqu'un argument est passé à un ensemble de clauses

$$p_1 \texttt{ -> } e_1 \mid \ldots \mid p_n \texttt{ -> } e_n$$

– la première clause qui filtre l'argument est exécutée, les autres sont ignorées.
– si aucune clause ne filtre l'argument, une exception est levée.

Filtrages incomplets

La définition est dite incomplète lorsqu'il existe une expression qui n'est filtrée par aucun motif. Dans ce cas, un message d'avertissement est indiqué à la compilation.

```
let simple x = match x with Carte{figure = Simple k} -> k
```

```
Characters 15-58:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Joker
  let simple x = match x with Carte {figure = Simple k} -> k
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
val simple : carte -> int = <fun>
```

Usage : Il est préférable d'éviter les définitions incomplètes.

Slide 52

Les motifs (patterns)

Ils peuvent :
– être emboîtés, contenir des produits et des sommes
– ne pas énumérer tous les champs des enregistrements
– lier une valeur intermédiaire avec mot clé `as`
– n'imposer aucune contrainte (représentés par le caractère `_`)
Une variable ne peut pas être liée deux fois dans le même motif.

Slide 53

```
let egal x y = match x, y with z, z -> true | _ -> false;;
```

```
Characters 34-35:
  let egal x y = match x, y with z, z -> true | _ -> false;;
                                    ^
This variable is bound several times in this matching
```

Exercices

Exercise 8 (Jeu de cartes) *Deux cartes sont incompatibles si aucune n'est le Joker et si elles sont différentes. Elles sont compatibles sinon. Un ensemble de cartes dont toutes les paires de cartes sont compatibles est compatible. Les ensemble de cartes sont représenté par des listes, mais considérés à permutation près.*

Écrire une fonction `compatibles` *qui prend un ensemble de cartes et retourne l'ensemble de toutes les sous parties compatibles les plus larges possibles.* Answer

En déduire une fonction testant si une main contient un carré (au moins autre cartes compatibles). Answer □

Slide 54

Exercise 9 (La fonction Sigma) *1. Écrire une fonction* `sigma : ('a -> int) -> 'a list -> int` *telle que l'expression* `sigma f l` *calcule la formule* $\sum_{x \in 1} f(x)$. Answer

2. Écrire la fonction `pi : ('a -> int) -> 'a list -> int` *analogue mais en remplaçant la somme par le produit.* Answer

3. Comment généraliser la fonction, de façon à partager le code en commun, en une fonction

```
fold : (int -> int -> int) ->
       int -> ('a -> int) -> 'a list -> int
```

Donner deux versions de ***fold****, d'abord une seule fonction globale récursive, puis utiliser une fonction locale récursive auxiliaire en lui passant le minimum d'argument(s).* Answer

4. *Réécrire la fonction* sigma *en utilisant une fonction de librairie.* Answer

Slide 55

□

Les exceptions (exemple)

Slide 56

```
exception Perdu

let rec cherche une_aiguille botte_de_paille =
  match botte_de_paille with
    brin :: botte ->
      if une_aiguille brin then brin
      else cherche une_aiguille botte
  | [] ->
      raise Perdu
let k =
  try
    let name, number =
      cherche  (fun x -> fst x =  "Louis")
        [ "Georges", 14; "Louis", 5; ] in number
  with Perdu -> 10;;
```

Exercise 10 (Exceptions) *Écrire un programme analogue sans utiliser d'exception.* Answer □

Les exceptions (suite)

Syntaxe

– Définition (phrase)	`exception` C [`of` t]
– Levée (expression)	`raise` e
– Filtrage (expression)	`try` e `with` p_1 `->` $e_1 \ldots \mid p_n$ `->` e_n

Slide 57 Remarquer l'analogie avec la construction de filtrage.

Exceptions pré-définis

Exception	Usage	gravité
`Invalid_argument of string`	accès en dehors des bornes	forte
`Failure of string`	liste vide, retourne le nom de la fonction	moyenne
`Not_found of unit`	fonctions de recherche	nulle
`Match_failure of ...`	Échec de filtrage	fatale
`End_of_file`	Fin de fichier	nulle

Slide 58

Sémantique

Slide 59

- Le type `exn` est un type somme.
- La levée d'une exception arête l'évaluation en cours et retourne une valeur "exceptionnelle" (du type `exn`).
- Une exception ne peut être éventuellement rattrapée que si elle a été protégée par une expression `try` e_1 `with` m
  - si l'évaluation de e_1 retourne une valeur normale celle-ci est retournée sans passer par le filtre.
  - si l'évaluation de e_1 retourne une valeur exceptionnelle, alors celle-ci est passée au filtre m. Si une des clauses p_i `->` e_i filtre l'exception, alors e_i est évaluée, sinon l'exception n'est pas capturée, et la valeur exceptionnelle est propagée.
- On peut observer une exception (*i.e.* l'attraper et la relancer) :

```
try f x with Failure s as x -> prerr_string s; raise x
```

Usage des exceptions

Slide 60

Il est préférable de définir de nouvelles exceptions plutôt que de réutiliser les exceptions pré-définies afin d'éviter tout ambiguïté.

On utilise les exceptions pour

- reporter une erreur et interrompre le calcul en cours ;
  Par exemple, l'exception `Match_failure` est levée à l'exécution lorsqu'une valeur n'est pas filtrée. Elle indique le nom du fichier et la position filtre en cause.
- communiquer une valeur de retour rare ou particulière ;
  Par exemple, les exceptions `Not_found` et `End_of_file` sont utilisées pour programmer et les lever est un comportement tout à fait normal.

Exemple : la commande Unix `cat`

Une version restreinte :

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

Slide 61

Exercise 11 (La commande cat) *Écrire la version complète, qui concatène l'ensemble des fichiers dont les noms sont passés en argument ou bien reproduit le flux d'entrée lorsque la commande est appelée sans argument.* Answer □

Exercise 12 (La commande grep) *Écire la commande unix `/bin/grep` sur le modèle de la commande `cat` et en utilisant la librairie `Str`.* Pour compiler, on il faut utiliser la commande :

```
ocamlc -o grep str.cma grep.ml
```

ou bien la commande Makefile générique avec `LIBS=str`. Answer □

Exercise 13 (La commande agrep) *Un raffinement de la commande `grep` consiste à traiter les lignes par regions plutôt qu'individuellement. Par exemple,*

```
./agrep -d '^let ' try agrep.ml
```

permet de recherche les définitions de phrases (dans un fichier bien indenté) qui contiennent le mot clé. Le premier argument optionnel (sa valeur par défaut est `$` désignant la fin de ligne), ici `^let` est le délimiteur[a], le second argument, ici `try` est le motif à rechercher, et les arguments suivant sont des noms de fichiers.

Slide 62

Chaque ligne filtré par le délimiteur commence une nouvelle région qui continue jusqu'à la prochaine ligne (exclue) qui est également filtrée par le délimiteur. La commande `agrep` imprime sur la sortie standard chaque région dont une des lignes au moins est contient le motif.

Implémenter la commande `agrep`. (On essayera de préserver le comportement incrémental qui consiste à afficher une région dès que possible, sans attendre la fin du fichier.) Answer

[a]Les ' sont les quotation du shell, mais ne font pas partie de l'argument passé par le shell à `agrep`.

Une limitation est que le motif ne peut pas se superposer sur plusieurs lignes. Lever cette limitation. Answer

De même le délimiteur ne peut pas se superposer sur plusieurs lignes. Comment peut-on lever cette limitation ? Answer □

Slide 63

Exercices

Slide 64

Exercise 14 (La calculette) *- Écrire une fonction sigma qui prend une fonction* `f`*, un tableau et deux indices et qui calcule* $\sum_{i=j}^{k} f(t.(i))$

- En déduire le calcul de la somme des éléments, des carrés ou la moyenne des éléments d'un sous-tableau.

- Faire un exécutable à partir de l'exercice précédent. Les arguments sont reçus sur la ligne de commande.

- Par défaut le programme calcule la somme. Si le premier argument est la chaîne `-carre` *ou* `-moyenne`*, alors il calcule la somme ou la moyenne des éléments restants.* Answer

- Séparer le programme en deux fichier `sigma.ml` *pour le code de la fonction sigma et* `calculette.ml` *pour le code principal. Écrire un fichier d'interface* `sigma.mli` *permettant de compiler* `calculette.ml` *sans avoir à compiler* `sigma.ml`*.* Answer

- Écrire un `Makefile` *manuellement.* Answer

- Utiliser le Makefile générique □

Exercices (suite)

Exercise 15 (La commande Unix `wc`) *compte l'ensemble des caractères, mots et lignes de chacun des fichiers passés en arguments ainsi que les totaux respectifs et les imprimes dans la sortie standard.* Answer □

Slide 65

La trace (mode interactif)

Fonctionne seulement en mode interprété à l'aide de directives

Mise en œuvre

```
let observe x y = x + y;;
#trace observe;;
#untrace observe;;
#untrace_all;;
```

Slide 66

Changer l'imprimeur par défaut

La trace se combine souvent avec l'ajout d'imprimeurs

```
let imprimeur x = Printf.printf "%d0" x;;
#install_printer imprimeur;;
1;;
#remove_printer imprimeur;;
```

L'argument *`imprimeur`* est le nom d'une fonction de type τ `-> unit` avec laquelle les valeurs du types τ seront imprimées.

Le débogueur

Fonctionne seulement en mode compilé.

Permet de faire du pas à pas en avant **et en arrière**, de visualiser l'état de la pile et les valeurs des variables locales.

Mise en œuvre

Compiler avec `ocamlc -g`

Slide 67

Appeler `ocamldebug` *`nom-du-programme`*

Il existe une interface conviviale pour Emacs : appeler la commande `camldebug` (taper `ESC X camldebug` *`nom-du-programme`*)

Si le programme n'est pas trouvé, il faut
- ajuster le chemin d'accès (dans le shell avant de lancer emacs)
- passer le chemin absolu, par exemple sous caml debug, avec
  `set program nom-absolu-du-programme`

Si le programme prend des arguments `a1`, `a2`, *etc.* on peut les passer par

```
set arguments a1 a2
```

La trace arrière (mode batch)

Permet le débogage des exceptions non rattrapées de façon légères.

Compiler avec l'option `-g` comme pour le débogueur.

Lancer le programme (version bytecode) avec la variable d'environnement `OCAMLRUNPARAM` valant "b", soit en affectant cette variable locatement au lancement du programme :

Slide 68

```
OCAMLRUNPARAM=b ./mon_programme ses_arguments
```

ou en affectant cette variable globalement avant de lancer le programme.

```
export OCAMLRUNPARAM=b
./mon_programme ses_arguments
```

Polymorphisme

Le typeur infère les types d'un programme. Il existe souvent plusieurs solutions :

```
let identity x = x;;
```

```
: int -> int
: string -> string
: 'a * 'a -> 'a * 'a
: 'a -> 'a
```

Slide 69

Le typeur retourne le type le plus général (il en existe toujours un si le programme est typable).

```
let identity x = x;;
```

```
val identity : 'a -> 'a = <fun>
```

Les autres se retrouvent par instantiation des variables de type. On dit que l'`identity` est polymorphe.

Support du polymorphisme

La construction de liaison introduit le polymorphisme.

```
let apply f x = f x in apply succ 1, apply print_int 1;;
```

Limitations

- un argument de fonction n'est pas polymorphe :

  ```
  (fun apply -> apply succ 1, apply print_int 1)
          (fun f x -> f x);;
  ```

  Les deux occurrences de `apply` doivent avoir le même type. *(Sinon, on ne sait plus deviner le type de* `apply`*)*

- Un calcul (une application) n'est pas polymorphe :

  ```
  let f = identity identity in f 1, f true;;
  ```

  L'expression `identity identity` n'est pas polymorphe.
  Une application pourrait produire des effets de bords — lecture et écriture — avec des types différents :

  ```
  let v = ref [] in v := [ 1 ]; not (List.hd v);;
  ```

Slide 70

Variable de type “faible”

Slide 71

Elle indique un type particulier pas encore connu.

```
let r = ref [];;
```

```
val r : '_a list ref = {contents=[]}
```

La valeur `r` n'est pas polymorphe (c'est une application).

Une variable de type `'_a`, dite *faible*, n'est pas une variable polymorphe : elle représente un type qui sera fixé utérieurement.

```
r := [1]; r;;
```

```
- : int list ref = {contents=[1]}
```

```
r := [true];;
```

```
Characters 0-1:
This expression has type int list ref but is here used with
type bool list ref
```

Variables de type faible (suite)

Slide 72

Les variables peuvent se retrouver afaiblies involontairement :

```
let map_apply = List.map (fun (f,x) -> f x);;
```

```
val map_apply : (('_a -> '_b) * '_a) list -> '_b list = <fun>
```

perdant ainsi le polymorphisme.

Pour conserver le polymorphisme, il suffit de retarder la spécialisation de la fonction en passant un argument supplémentaire :

```
let map_apply l = List.map (fun (f,x) -> f x) l;;
```

```
val map_apply : (('a -> 'b) * 'a) list -> 'b list = <fun>
```

Cela ne change pas le sens lorsque la fonction attend de toute façon un argument supplémentaire.

Calculs gelés

L'application d'une fonction à un argument effectue le calcul immédiatement.

Il est parfois souhaitable de retarder le calcul
– Pour éviter un gros calcul inutile.
– Pour retarder ou répéter les effets de bord associé au calcul.

Slide 73 On peut contrôler le moment d'exécution des calculs à l'aide de l'abstraction et de l'application.

```
let plus_tard f x = fun () -> f x;;
let un = plus_tard print_int 3;;
let mark = plus_tard print_string "*";;
mark(); un(); mark();;
```

```
*3*- : unit = ()
```

Calculs gelés (suite)

Exercise 16 (Gelé les calculs) *On considère le programme :*

```
let ifz^^e9ro n sioui sinon = if n = 0 then sioui else sinon;;
ifz^^e9ro 3 (print_string "oui") (print_string "non");;
```

Expliquer l'anomalie.

Slide 74 *Corriger cette anomalie en redéfinissant la fonction* `ifz^^e9ro` *avec le type* `int -> (unit -> 'a) -> (unit -> 'a) -> 'a`*. On donnera le programme corrigé complet.* Answer □

Streams

Exercise 17 (Calculs paresseux) *Un stream est une liste potentiellement infinie. Elle doit bien sûr être représentée de façon finie, les calculs des s'effectuant paresseusement au fur et à mesure des besoins et mémorisés pour une relecture ultérieure. On peut le représenter par le type concret suivant :*

Slide 75

```
type 'a stream = 'a status ref
and 'a status =
    Nil
  | Cons of 'a * 'a stream
  | Exception of exn
  | Frozen of (unit -> ('a * 'a stream));;
```

Un stream dont le statut est une exception est un stream dont l'évaluation a lancé (et donc devra relancer) cette exception.

1. Définir les fonctions `hd`, `tl` *et* `nth` *sur les streams qui retournent le premier élément, le reste du stream, et le $n^{ième}$ élément. (On aura avantageusement factoriser le code de* `hd` *et* `tl` *en utilisant une fonction auxiliaire* `next`*.)*

Answer

Slide 76

Streams (suite de l'exercice)

Slide 77

2. Donnez également des fonctions de construction `nil`*,*
`cons : 'a -> 'a stream -> 'a stream` *et une fonction*
`freeze : (unit -> ('a * 'a stream)) -> 'a stream`*, qui retourne le stream vide, ajoute un élément explicite en tête d'un stream et construit un stream à partir d'une fonction génératrice.* Answer
Construire le stream des entiers pairs. Answer
3. Définir une fonction
`filter : ('a -> bool) -> 'a stream -> 'a stream` *qui prend un prédicat et un stream et retourne le stream des éléments qui satisfont le prédicat. En déduire le stream des entiers pairs à partir du stream des entiers.* Answer
4. En déduire une fonction `crible` *qui prend un stream d'entiers* `s` *et qui retourne le sous-stream des éléments qui ne sont pas divisibles par les éléments qui les précédent. En déduire le stream des entiers premiers.* Answer □

La programmation traditionnelle

Slide 78

Les programmes sont écrits de façon linéaire.

On peut seulement ajouter de nouvelles définitions à la fin du programme.

Réutilisation du code par "couper-coller".

Pas de partage de code :
– duplication du code, et des erreurs !
– difficulté de maintenance.
Pas de protection du code :
– Pas de types abstrait.
– Abstraction de valeur, seulement.

La programmation modulaire

Découpage du programme en morceaux compilables indépendamment

Rendre les gros programmes compilables

Donner de la structure au programme

Rendre les gros programmes compréhensibles

Spécifier les liens (interfaces) entre les composantes

Slide 79

Rendre les gros programmes maintenables

Identifier des sous-composantes indépendantes

Rendre les composantes réutilisables

Deux approches

Programmation par objets

Les objets sont définis à partir de classes.

Les classes peuvent être construites à partir d'autres classes par héritage.

Un langage de module très riche

Slide 80

Les modules de bases permutent de contrôler la visibilité des champs en cachant certains définitions de valeurs ou de types.

Les foncteurs sont des fonctions des modules vers les modules, permettant de générer de nouveaux modules à partir d'autres.

Les modules sont également le support du mécanisme de compilation séparée.

Modules et compilation séparée

Slide 81

Une unité de compilation A se compose de deux fichiers :
- Le fichier d'implémentation `a.ml` :
  une suite de phrases.
- Le fichier d'interface `a.mli` (optionnel) :
  une suite de spécifications.

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.

Les spécifications sont (essentiellement) :

spécification de valeurs	`val` $x : \sigma$
spécification de types abstraits	`type` t
spécification de types manifestes	`type` $t = \tau$
spécification d'exceptions	`exception` E

Compilation séparée d'un programme

Slide 82

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	

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

Utilisation de la commande `make`

La commande `make` permet de maintenir à jour une application composée de plusieurs composantes, indépendemment du langage (C, tex, ocaml, etc.).

Elle utilise un fichier `Makefile` (par défaut dans le directory courant) qui décrit les fichiers sources, les fichiers à fabriquer, la façon de les fabriquer, ainsi que les dépendences.

Slide 83

Un Makefile spécifique

On peut écrire un fichier Makefile spécialisé pour chaque application. Par exemple, celui-ci fonctionne pour l'ensemble des exercices de ce cours.

Le Makefile générique

Le Makefile générique pour OCaml, écrit par Markus Mottl, couvre la plupart des situations.

- Copier `Makefile` dans votre dossier de travail.
- Éditer les variables `SOURCES` et `RESULT`

  (au plus un exécutable à la fois)
- Faire `make`

Mode graphique Il suffit d'ajouter la définition suivante :

```
LIBS= graphics
```

Voir README pour en savoir plus sur les autres commandes.

Voir README pour en savoir plus

Slide 84

Modules.

Les modules en quelques mots

Slide 85

- Un petit langage fonctionnel typé
- pour manipuler des collections de définitions (de valeurs, de types) du langage de base.
- Leurs types : des collections de déclarations/spécifications (types des valeurs, déclarations des types).
- Modules emboîtés.
- Fonctions (foncteurs) et application de fonctions.

Largement indépendant du langage de base.

Modules de base

Les structures sont collections de phrases

`struct` $p_1 \ldots p_n$ `end`

Les phrases sont celles du langage de base, plus

définition de sous-modules	`module` $X = \ldots$
définition de type de module	`module type` $S = \ldots$

Slide 86

Le nommage d'un module se fait à l'aide de la liaison `module`.

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

Utilisation d'un module

On fait référence aux composantes d'un module avec la notation pointée *module*`.`*composante*

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

Autre possibilité : la directive `open` *module* permet d'omettre le préfixe et le point :

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

Slide 87

Exemples :

List.map

Hastbl.find

Pervasives.exit

Modules emboîtés

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;;
```

Slide 88

La notation pointée et la construction `open` s'étendent naturellement aux sous-modules :

```
module Q =
  struct
    let z = T.R.x
    open T.R
      ...
  end
```

NB : La notation `open T.R` rend les composantes de `T.R` visibles dans la suite du corps de `Q` mais n'ajoute pas ces composantes à `Q`.

Les types des modules de base

Les *signatures* : collections de spécifications (de types).

`sig` *spécification* ...*spécification* `end`

Slide 89

spécification de valeurs	`val x :` σ
spécification de types abstraits	`type` t
spécification de types manifestes	`type` $t = \tau$
spécification d'exceptions	`exception` E
spécification de sous-modules	`module` $X : M$
spécification de type de module	`module type` $S\ [= M]$

Nommage d'un type de module : par la liaison `module type`

```
module type MA_SIGNATURE = sig ... end
```

Synthèse de signature

Le système infère les signatures des modules (comme il infère les types des valeurs).

Slide 90

Module

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

Signature inférée

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

Restriction par une signature

La construction `(`*structure* `:` *signature*`)`
- vérifie que la structure satisfait la signature (toutes les composantes spécifiées dans la signature doivent être définies dans la structure, avec des types au moins aussi généraux) ;
- 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 =` ...

Slide 91

Sucre syntaxique :	est équivalent à :
`module X : S = M`	`module X = (M : S)`

Utilisée pour masquer des composantes et des types.

Restriction (Écritures équivalentes)

Slide 92

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

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

```
M.x;;            (* ``S.x is unbound'' *)
M.y + 1;;        (* ``S.y is not of type int'' *)
```

Vues isomorphes incompatibles

Il est parfois important de distinguer des types isomorphes. Par exemples Euros et Dollars sont tous deux représentés par des flottants. Pourtant, il ne faut pas les confondre.

Ce sont deux espaces vectoriels, isomorphes mais disjoints, avec pour unités respectives l'`euro` et le `dollar`.

Slide 93

```
module Float =
  struct
    type t = float
    let un = 1.0
    let plus = (+.)
    let prod = ( *. )
  end;;
```

```
module type MONNAIE =
  sig
    type t
    val un   : t
    val plus : t -> t -> t
    val prod : float -> t -> t
  end;;
```

La multiplication devient une opération externe sur les flottants.

Monnaies incompatibles

Dans `Float` le type `t` est concret donc il peut être confondu avec "float". Par contre, il est abstrait dans les modules `Euro` et `Dollar` ci-dessous :

```
module Euro = (Float : MONNAIE);;
module Dollar = (Float : MONNAIE);;
```

Les types `Euro.t` et `Dollar.t` sont isomorphes mais incompatibles.

Slide 94

```
let euro x = Euro.prod x Euro.un;;
Euro.plus (euro 10.0) (euro 20.0);;
Euro.plus (euro 50.0) Dollar.un;;        Erreur
```

Pas de duplication de code entre `Euro` et `Dollar`.

Vues multiples d'un même module

On peut donner une interface restreinte pour, dans un certain contexte, ne permettre que certaines opérations (typiquement, interdire la création de valeurs) :

Slide 95

```
module type PLUS =
  sig
    type t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS)
```

```
module type PLUS_Euro =
  sig
    type t = Euro.t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS_Euro)
```

À gauche, le type `Plus.t` est incompatible avec `Euro.t`.

À droite, le type `t` est partiellement abstrait et compatible avec "Euro.t", et la vue `Plus` permet de manipuler les valeurs construites avec la vue `Euro`.

La notation `with`

La notation `with` permet d'ajouter des égalités de types dans une signature existante.

L'expression `PLUS with type t = Euro.t` est une abréviation pour la signature

Slide 96

```
sig
  type t = Euro.t
  val plus: t -> t -> t
end
```

On peut alors écrire

```
module Plus = (Euro : PLUS with type t = Euro.t);;
Plus.plus Euro.un Euro.un;;
```

Elle permet de créer facilement des signatures partiellement abstraites.

Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules :

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

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module *S*. Il fait référence aux composantes de son paramètre avec la notation pointée.

Slide 97

```
module F = functor(X : S) ->
  struct
    type u = X.t * X.t
    let y = X.g X.v
  end;;
```

Utilisation des modules paramétrés

Slide 98

Définir des structures qui dépendent d'autres

– Les librairies Map, Set, Hashtbl, *etc.* dépendent d'une fonction de comparaison.
  Le type des éléments avec leur fonction de comparaison :

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end;;
```

  Le type des ensembles

```
module type S = sig .. end
```

  Le foncteur qui "fabrique" des implémentations :

```
module Make:
  functor (Ord : OrderedType) -> S with type elt = Ord.t
```

– Les polynômes sont définis par rapport à une structure d'anneau sur les coefficients.

Application de foncteur

Slide 99

On ne peut pas directement accéder au corps d'un foncteur. Il faut au préalable l'appliquer explicitement à une implémentation de la signature de son argument (tout comme on applique une fonction ordinaire).

```
module P1 = F(M1)
module P2 = F(M2)
```

`P1`, `P2` s'utilisent alors comme des structures ordinaires :

```
(P1.y : P2.u)
```

`P1` et `P2` partagent entièrement leur code.

Exemple : les ensembles d'entiers

```
module IntSet =
  Set.Make
    (struct type t = int let compare = compare end);;
```

Modules et compilation séparée

Fichiers sources : `a.ml`, `a.mli`, `b.ml` `B` utilse `A`.

Le programme se comporte comme le code monolithique :

```
module A =
  (struct (* contenu de a.ml *) end
    : sig (* contenu de a.mli *) end)
module B = struct (* contenu de b.ml *) end
```

Slide 100

Exercice

Exercise 18 (Polynômes à une variable (***))

– *Réaliser une bibliothèque fournissant les opérations sur les polynômes à une variable. Les coefficients forment un anneau passé en argument à la bibliothèque.* Answer

– *Utiliser la bibliothèque pour vérifier par exemple l'égalité* $(1+X)(1-X) = 1-X^2$. Answer

Slide 101

– *Vérifier l'égalité* $(X+Y)(X-Y) = (X^2-Y^2)$ *en considérant les polynômes à deux variables comme polynôme en* X *à coefficients les polynôme en* Y.

– *Écrire un programme qui prend un polynôme sur la ligne de commande et l'évalue en chacun des points lus dans* `stdin` *(un entier par ligne) et imprime le résultat dans* `stdout`. Answer

– *Utiliser la commande make pour compiler le programme.*

□

Exercices.

Menu conseillé

Slide 102

1. **Mise en œuvre**
   - C'est un point de passage obligé
     (exercices 1, 2 et 3 du menu).
   - Faire ces exercices en regardant les solutions.
2. **Petits exercices**
   - On peut les mettre au point “en mode calculette”
     (en les écrivant dans un fichier sous emacs et en exécutant les commandes au fur et à mesure).
   - Faire les exercices plus conséquents en mode compilé.
   - En faire au moins deux sans avoir regardé la solution.
3. **Choisir un exercice plus conséquent**
   - En choisir un et le faire sans regarder la solution.
   - En cas d'échec, refaire de petits exercices...
4. **En dernier**
   - Un exercice plus difficile, par exemple, sur les polynômes.

Quelques tuyaux

Slide 103

Relire le cours de façon interactive

- en cliquant sur tous les pointeurs disponibles (et récursivement à la profondeur qui vous plaira).
  Même si vous ne lisez que les premières lignes des documents pointez, vous connaitrez au moins leur existence.
- en faisant tous les exercices du cours au fur et à mesure.

Gardez le cours et la documentation sous la main

Si possible en parallèle une fenêtre de votre brouteur et une fenêtre emacs sur le même écran : on programme avec la documentation ou des exemples sous les yeux.

A Exercices

Calculs numériques

Fonctions récursives, simple ou multiples, avec calculs sur les nombres.

Exercise 19 (Fibonnaci) *Écrire, en mode intéractif, la fonction de fibonnaci définie par induction*

$$fib(n) = \begin{cases} 1 & \textit{if } n = 0 \textit{ ou } n = 1 \\ fib(n-1) + fib(n-2) & \textit{otherwise} \end{cases}$$

Answer

Fabriquer un exécutable qui prend son argument sur la ligne de commande. Answer □

Exercise 20 (Nombres complexes) *Définir un type `complex` pour représenter les nombres complexes en coordonnées cartésiennes (on choisira des coefficinets flottants).* Answer
Définir l'addition `++` et la multiplication `**` sur les nombres complexes. Answer
En déduire la calcul de $(z_n) = (1 + i/10)^n$ et la valeur de z^{99}. Answer
Donner le calcul à l'aide de la fonction `power` définie ci-dessus. Answer □

Les librairies

Exercise 21 (Listes) *Consulter la librairie List.*
Implémenter la fonction `length` Answer
La fonction est-elle tail-récursive ? (Donner, le cas échéant, une version fonctionnelle (qui n'utilise pas de référence) en récursion terminale. Answer
Implémenter la fonction `find`. Answer
Implémenter d'autres fonctions de la librairie... □

Exercise 22 *Consulter et comparer les librairies Array et String.*

On se propose de réimplémenter les opérations sur les tableaux en utilisant seulement les fonctions `length` *et* `make`.

Donner les opérations `get` *et* `set`. Answer

Impémenter l'opération `init` Answer

Implémenter l'opération `make_matrix` *;;* Answer

Implémenter l'opération `blit` *:* Answer

Comparer la librairie String avec la librairie Array. □

Exercise 23 (Les libairies) *Consulter la librairie Hashtbl.*

Implémenter (une version simplifiée du module) de ce module, en utilisant l'égalité générique comme comparaison. (On ne s'autorise que la fonction `hash` *du module* `Hashtbl`*).*

Pourquoi y a-t-il une également une interface fonctionnelle ? Answer

Donner un exemple d'utilisation de la version fonctorisée. Answer □

B Answers to exercises

Exercise 1, page 17

—Il affiche les noms de fichiers du répertoire courant donc le nom se termine par `.ml`.

Exercise 2, page 26

L'expression `f (x y)` est intruse. Les autres sont équivalentes.

Exercise 3, page 27

La seconde (dernière) expression retarde l'évaluation jusqu'à ce que le premier (les deux) argument(s) soi(ent) fourni(s). Cela peut aussi délaisser ou dupliquer certains calculs :

```
let f x = print_int x; fun y -> x + y;;
let test f = let f1 = f 1 in f1 2 + f1 3;;
test f;;
```

1− : int = 7

```
test (fun x y -> f x y);;
```

11− : int = 7

Exercise 4, page 33

Exécuter le programme pour voir !

La troisième est imposée par la norme IEEE. (Notez que `compare nan nan` retourne bien 0.)

Exercise 5, page 34

```
let last = Array.length Sys.argv - 1 in
let print_args first =
  (* no white space before the first argument *)
  if first <= last then print_string Sys.argv.(first);
  for i = first + 1 to last
  do print_char ' '; print_string Sys.argv.(i) done in
if last > 0 && Sys.argv.(1) = "-n" then print_args 2
else
  begin
    print_args 1;
    print_newline()
  end;;
```

Exercise 6, page 36

true.ml
```
exit 0;;
```
true.ml

false.ml
```
exit 1;;
```
false.ml

Pour les compiler :
```
ocamlc -o true true.ml
ocamlc -o false false.ml
```
Pour les tester :
```
./true; echo $?
./false; echo $?
```

Exercise 7, page 47

```
let rec print_double f x =
  match x with
    Vide -> ()
  | Gauche (x, d) -> f x; print_double f d
  | Droite (d, x) -> print_double f d; f x;;
print_double print_int d;;
```

1234− : unit = ()

Exercise 7 (continued)

```
let list_of_double l =
  let d^^e9j^^e0_construite = ref [] in
  let accumule x = d^^e9j^^e0_construite := x :: !d^^e9j^^e0_construite in
  print_double accumule l;
  List.rev !d^^e9j^^e0_construite;;
let l = list_of_double d;;
```

val l : int list = [1; 2; 3; 4]

Exercise 7 (continued)

On utilise une fonction auxilliaire `d^^e9double` qui prend en argument supplémentaire le morceau de (vraie) liste déjà construite.
```
let rec d^^e9double d^^e9j^^e0_construite ^^e0_d^^e9doubler =
  match ^^e0_d^^e9doubler with
  | Vide -> d^^e9j^^e0_construite
  | Gauche (h,t) -> h :: (d^^e9double d^^e9j^^e0_construite t)
  | Droite (t, h) -> d^^e9double (h :: d^^e9j^^e0_construite) t;;

let list_of_double l = d^^e9double [] l;;
let l = list_of_double d;;
```

Exercise 8, page 54

La seule difficulté provient de la carte `Joker`, qui peut remplacer n'importe quelle autre carte. Un joker est compatibles avec deux cartes qui ne sont pas compatibles entre elles. La relation de compatibilité n'est donc pas une relation d'équivalence.

On définit la relation asymétrique binaire `rempla^^e7able_par` : la carte `Roi` est remplaçable par la carte `Joker`, mais pas l'inverse.
```
let remplacable_par x y =
  match x, y with
    Carte u, Carte v -> u.figure = v.figure
  | _, Joker -> true
  | Joker, Carte _ -> false
```

```
let non_remplacable_par x y = not (remplacable_par y x);;
```

La fonction `compatibles` est définie pas induction. Si la main est vide, l'ensemble des parties compatibles est vide. Si le premier élément de la main est un joker, alors les parties compatibles sont celles du reste de la main dans chacune desquelles il faut ajouter un joker. Sinon, le premier élément est une carte ordinaire : les parties compatibles sont d'une part l'ensemble des cartes compatibles avec celle-ci, et l'ensemble des parties compatibles avec les cartes qui ne sont pas compatibles avec celle-ci.

```
let rec compatibles carte =
  match carte with
  | Joker :: t ->
      List.map (fun p -> Joker::p) (compatibles t)
  | h :: t ->
      let found = h :: List.find_all (rempla^^e7able_par h) t in
      let rest = compatibles (List.find_all (non_rempla^^e7able_par h) t) in
      found :: rest
  | [] -> [];;
```

Exercise 8 (continued)

```
let carr^^e9 main =
  List.exists (fun x -> List.length x >= 4) (compatibles main);;
```

Exercise 1, page 54

```
let rec sigma f l =
  match l with
  | [] -> 0
  | x :: rest -> f x + sigma f rest;;
```

Exercise 2, page 54

```
let rec pi f l =
  match l with
  | [] -> 1
  | x :: rest -> f x * pi f rest;;
```

Exercise 3, page 55

```
let rec fold (oper : int -> int -> int) (x0 : int) f x =
  match x with
  | [] -> x0
  | h :: rest -> oper (f h) (fold oper x0 f rest);;
let fold (oper : int -> int -> int) (x0 : int) f =
  let rec aux x =
    match x with
    | [] -> x0
    | h :: rest -> oper (f h) (aux rest) in
  aux;;
let sigma f l = fold ( + ) 0 f l;;
let pi f l = fold ( * ) 1 f l;;
```

Exercise 4, page 55

```
let sigma f l = List.fold_right (fun x r -> (f x) + r) l 0;;
```

Exercise 10, page 56

```
type 'a valeur = Trouve of 'a | Perdu
let rec cherche aiguille botte_de_paille =
  match p with
    brin :: botte ->
```

```
      if aiguille brin then Trouve brin
      else cherche aiguille botte
  | [] -> Perdu
let k =
  match cherche (fun x -> fsx x = "Georges")
           [ "Louis", 14; "Georges", 5;] with
  | Trouve x -> x
  | Perdu -> 10
```

Exercise 11, page 61

cat.ml

```
let echo chan =
  try while true do print_string (input_line chan); print_newline() done
  with End_of_file -> ();;

if Array.length Sys.argv <= 1 then echo stdin
else
  for i = 1 to Array.length Sys.argv - 1
  do
    let chan = open_in Sys.argv.(i) in
    echo chan;
    close_in chan
  done;;
```

cat.ml

Exercise 12, page 61

grep.ml

```
let pattern =
  if Array.length Sys.argv < 2 then
    begin
      print_endline "Usage: grep REGEXP file1 .. file2";
      exit 1
    end
  else
    Str.regexp Sys.argv.(1);;

let process_line l =
  try let _ = Str.search_forward pattern l 0 in print_endline l
  with Not_found -> ()

let process_chan c =
  try while true do process_line (input_line c) done
  with End_of_file -> ();;

let process_file f =
  let c = open_in f in process_chan c; close_in c;;

let () =
  if Array.length Sys.argv > 2 then
    for i = 2 to Array.length Sys.argv - 1
    do process_file  Sys.argv.(i) done
  else
    process_chan stdin;;
```

grep.ml

Exercise 1, page 62

agrep.ml

```
(* On parse  les arguments sur la ligne de commande
   en utilisant le module Arg *)

let split_string = ref "$";;
let set_split s = split_string := s;;

let pattern_string = ref None
let interactive = ref true
let files = ref []
let add_pattern_or_file s =
  match !pattern_string with
    None -> pattern_string := Some s
  | _ -> files := s :: !files;;

let options =
  [ "-d", Arg.String set_split, "Set delimiter pattern"; ]
let usage = "Usage: grep [-d DELIMITER_STRING ] REGEXP file1 .. filen";;
let () =   Arg.parse options add_pattern_or_file usage;;

let delim = Str.regexp !split_string
let pattern =
  match !pattern_string with
  | None -> print_endline usage; exit 1
  | Some s -> Str.regexp s

let matches regexp line =
  try let _ = Str.search_forward regexp line 0 in true
  with Not_found -> false;;

(* le traitement d'une region *)
let process_region lines =
  if List.exists (matches pattern) lines then
    begin
      List.iter print_endline (List.rev lines);
      flush stdout
    end;;

(* On accumule les lignes jusqu'a une ligne separatrice *)
let rec iter_region chan r =
  try
    let line = input_line chan in
    if matches delim line then
      (process_region r; iter_region chan [ line ])
    else iter_region chan (line :: r)
  with
    End_of_file -> process_region r;;

let process_chan chan = iter_region  chan []

(* main *)
let () =
  match !files with
    [] ->
      process_chan stdin
  | l ->
      let process file =
        Printf.printf "<<%d>>\n"
          (match !pattern_string with Some s -> Char.code s.[2] | _ -> 0);
        let c = open_in file in
        process_chan c;
        close_in c in
      List.iter process l;;
```

agrep.ml

Exercise 1 (continued)

Une solution simple consite redéfinir la fonction `process_region` :

```
let process_region lines =
  let region = String.concat "\n" (List.rev lines) in
  if matches pattern region then
    begin
      print_endline region;
      flush stdout
    end;;
```

Exercise 1 (continued)

C'est plus embêtant, sauf à abandonner le mode incrémental (on peut alors lire tout le fichier dans une chaîne de caractères dans un premier temps) ou à imposer une limitation sur le nombre de caractères ou de retours à la ligne filtrés par le délimiteur.

Exercise 14, page 64

```
open Sigma
let last = Array.length Sys.argv - 1;;
if last < 1 then exit 0;;
match Sys.argv.(1) with
| "-carre" ->
    let f x = let n = int_of_string x in n * n in
    print_int (sigma f Sys.argv 2 last)
| "-moyenne" ->
    let r = sigma int_of_string Sys.argv 2 last in
    print_float (float r /. float (Array.length Sys.argv - 2))
| _ ->
    print_int (sigma int_of_string Sys.argv 1 last)
;;
print_newline();;
% ocamlc -o calculette calculette.ml
% calculette -carre 1 2 3 4 5 6
91
```

Exercise 14 (continued)

sigma.ml

```
let sigma f t j k =
  let y = ref 0 in
  for i = j to k do y := !y + f t.(i) done;
  !y;;
```

sigma.ml

sigma.mli

```
val sigma : ('a -> int) -> 'a array -> int -> int -> int
```

sigma.mli

calculette.ml

```
open Sigma
let last = Array.length Sys.argv - 1;;
if last < 1 then exit 0;;
match Sys.argv.(1) with
```

```
| "-carre" ->
    let f x = let n = int_of_string x in n * n in
    print_int (sigma f Sys.argv 2 last)
| "-moyenne" ->
    let r = sigma int_of_string Sys.argv 2 last in
    print_float (float r /. float (Array.length Sys.argv - 2))
| _ ->
    print_int (sigma int_of_string Sys.argv 1 last)
;;
print_newline();;
```
calculette.ml

Exercise 14 (continued)

Makefile

Exercise 15, page 65

wc.ml
```
type counts = {
 mutable chars : int;
 mutable lines : int;
 mutable words : int;
};;

let wc_count = {chars = 0; lines = 0; words = 0};;
let wc_count_total = {chars = 0; lines = 0; words = 0};;

let reset_count () =
 wc_count.chars <- 0;
 wc_count.lines <- 0;
 wc_count.words <- 0;;

let cumulate () =
 wc_count_total.chars <- wc_count_total.chars + wc_count.chars;
 wc_count_total.lines <- wc_count_total.lines + wc_count.lines;
 wc_count_total.words <- wc_count_total.words + wc_count.words;;

let rec counter ic iw =
 let c = input_char ic in
 wc_count.chars <- wc_count.chars + 1;
 match c with
 | ' ' | '\t' ->
    if iw then wc_count.words <- wc_count.words + 1 else ();
    counter ic false
 | '\n' ->
    wc_count.lines <- wc_count.lines + 1;
    if iw then wc_count.words <- wc_count.words + 1 else ();
    counter ic false
 | c ->
    counter ic true;;

let count_channel ic =
 reset_count ();
 try counter ic false with
 | End_of_file -> cumulate (); close_in ic;;

let output_results s wc =
  Printf.printf "%7d%8d%9d %s\n" wc.lines  wc.words wc.chars  s
;;
```

```
let count_file file_name =
 try
  let ic = open_in file_name in
  count_channel ic;
  output_results file_name wc_count;
 with Sys_error s -> print_string s; print_newline(); exit 2
;;

let main () =
  let nb_files = Array.length Sys.argv - 1 in
  if nb_files > 0 then
    begin
      for i = 1 to nb_files do
        count_file Sys.argv.(i)
      done;
      if nb_files > 1 then output_results "total" wc_count_total;
    end
  else
    begin
      count_channel stdin;
      output_results "" wc_count;
    end;
  exit 0;;

main();;
```

wc.ml

Exercise 16, page 74

La fonction `ifz^^e9ro` évalue ses deux arguments (de plus l'ordre d'évaluation n'est pas déterminé).

Il faudrait que ses arguments soient systématiquement gelés, donc passés comme des fonctions de type `unit -> 'a`.

```
let ifz^^e9ro n sioui sinon = if n = 0 then sioui() else sinon();;
ifz^^e9ro 3 (fun() -> print_string "oui") (fun() -> print_string "non");;
```

Exercise 17, page 75

```
let next x =
  match !x with
  | Nil -> failwith "next"
  | Cons (h,t) -> h, t
  | Exception x -> raise x
  | Frozen f ->
      try let h, t as ht = f() in x := Cons (h, t); ht
      with exn -> x := Exception exn; raise exn;;

let hd x = fst (next x);;
let tl x = snd (next x);;

let rec nth l n =
  if n < 0 then failwith "nth"
  else if n = 0 then hd l
  else nth (tl l) (pred n);;
```

Exercise 17 (continued)

```
let nil = ref Nil
let cons x t : 'a stream = ref (Cons (x, t))
let freeze f : 'a stream = ref (Frozen f);;
```

Exercise 17 (continued)

```
let entiers_pair = let rec s n = freeze (fun () -> n, s (n + 2)) in s 0;;
nth entiers_pair 3;;
```

Exercise 17 (continued)

```
let rec filter f s =
  let rec first s =
    let h, t = next s in
    if f h then h, filter f t else first t in
  freeze (fun () -> first s);;
let entiers = let rec s n = freeze (fun () -> n, s (n + 1)) in s 0;;
let entiers_pairs = filter (fun x -> x mod 2 = 0) entiers;;
    nth entiers 3;;
    nth entiers_pairs 3;;
```

Exercise 17 (continued)

```
let rec crible s =
  let rec first s =
    let h, t = next s in
    let t' = filter (fun x -> x mod h <> 0) t in
    h, crible t' in
  freeze (fun () -> first s);;

let premiers = crible (tl (tl entiers));;
    nth premiers 100;;
```

Exercise 18, page 101

polynome.mli

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t                            (* type des éléments de l'anneau *)
    val zéro : t
    val unité : t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val print : t -> unit
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
module type POLYNOME =
  sig
    type c
        (* type des coefficients *)
    type t
        (* type des polynômes *)
    val zéro : t
    val unité : t
        (* unité pour le produit des polymôme.
           Superflue, car c'est un cas particulier de monôme, mais cela
           permet à la structure de polynôme d'être une superstructure de
           celles des anneaux *)
    val monôme : c -> int -> t
        (* monôme a k retourne le monôme a X^k *)
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
```

```
    val print : t -> unit
    val eval : t -> c -> c
  end
      ;;


(* Le fonction qui étant donné une structure d'anneau retourne une structure
de polynôme. Il faut faire attention à réexporter le types des coefficients
de façon partiellement abstraite afin que les opérations sur les coefficients
des polynômes puissent être manipulés de l'extérieur *)


module Make (A : ANNEAU) : (POLYNOME with type c = A.t)
    ;;
```

polynome.mli

polynome.ml

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t
    val zéro : t
    val unité : t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val print : t -> unit
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
module type POLYNOME =
  sig
    type c
    type t
    val zéro : t
    val unité : t
    val monôme : c -> int -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val print : t -> unit
    val eval : t -> c -> c
  end
      ;;

module Make (A : ANNEAU) =
  struct
    type c = A.t

    (* un monome est un coeficient et une puissance *)
    type monôme = (c * int)
    (* un polynome  est une liste de monomes triés par rapport au puissance *)

    (* il est essentiel de préserver cet invaraint dans le code ci-dessous *)
    type t = monôme list

    (* pour que la représentation soit canonique, on élimine aussi
       les coeficients nuls, en particulier le monome nul est la liste vide *)
    let zéro = []
```

```ocaml
(* on peut donc (grâce à l'invariant utiliser l'égalité générique *)
let rec equal p1 p2 =
  match p1, p2 with
  | [],[] -> true
  | (a1, k1)::q1, (a2, k2)::q2 -> k1 = k2 && A.equal a1 a2 && equal q1 q2
  | _ -> false

(* création d'un monôme *)
let monôme a k =
  if k < 0 then failwith "monôme: exposant négatif"
  else if A.equal a A.zéro then [] else  [a, k]

(* un cas particulier l'unité *)
let unité = [A.unité, 0]

(* attention à respecter l'invariant et ordonner les monômes *)
let rec plus u v =
  match u, v with
    (x1, k1)::r1 as p1, ((x2, k2)::r2  as p2) ->
      if k1 < k2 then
        (x1, k1):: (plus r1 p2)
      else if k1 = k2 then
        let x = A.plus x1 x2 in
        if A.equal x A.zéro then plus r1 r2
        else (A.plus x1 x2, k1):: (plus r1 r2)
      else
        (x2, k2):: (plus p1 r2)
  | [], _  -> v
  | _ , [] -> u

(* on pourrait faire beaucoup mieux pour éviter de recalculer les
   puissances de $k$, soit directement, soit en utilisant une
   mémo-fonction *)

let rec fois (a, k) =  (* on suppose  a <> zéro *)
  function
    | [] -> []
    | (a1, k1)::q ->
        let a2 = A.mult a a1 in
        if A.equal a2 A.zéro then fois (a,k) q
        else (a2, k + k1) :: fois (a,k) q

let mult p = List.fold_left (fun r m -> plus r (fois m p)) zéro

(* une impression grossière *)
let print p =
  List.iter
    (fun (a,k) ->
      Printf.printf "+ (";
      A.print a;
      Printf.printf ") X^%d " k)
    p

(* Puissance c^k, c un coefficient, k un entier >= 0.  Par dichotomie. *)
let rec puis c = function
  | 0 -> A.unité
  | 1 -> c
  | k ->
      let l = puis c (k lsr 1) in
      let l2 = A.mult l l in
        if k land 1 = 0 then l2 else A.mult c l2

let eval p c = match List.rev p with
```

```
      | [] -> A.zéro
      | (h::t) ->
         let (* Réduire deux monômes en un.  NB: on a k >= l. *)
             dmeu (a, k) (b, l) =
           A.plus (A.mult (puis c (k-l)) a) b, l
         in
         let a, k =  List.fold_left dmeu h t in
         A.mult (puis c k) a

  end
    ;;
```

polynome.ml

Exercise 18 (continued)

polytest.ml

```
(* pour charger le fichier compilé polynome.cmo dans le mode interactif *)

#load "polynome.cmo";;
open Polynome;;

(* l'anneau des entiers *)
module I =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let plus = ( + )
    let mult = ( * )
    let equal = ( = )
    let print = print_int
end;;

(* les polynômes à coefficients entiers *)
module X = Make (I);;

let xm = X.monôme;;
let p = X.mult (X.plus (xm 1 0) (xm 1 1)) (X.plus (xm 1 0) (xm (-1) 1));;
let q = X.plus (xm 1 0) (xm (-1) 2);;
X.equal p q;;

module Y = Make (X);;

let ym = Y.monôme;;
let x = ym (xm 1 1) 0;;                   (*  (1 X^1) Y^0   *)
let y = ym (xm 1 0) 1;;                   (*  (1 X^0) Y^1   *)
let moins_unité = ym (xm (-1) 0) 0;;      (*  ((-1) X^0) Y^0 *)
let moins p q = Y.plus p (Y.mult moins_unité q);;

let p = Y.mult (Y.plus x y) (moins x y);;
let q = moins (Y.mult x x) (Y.mult y y);;

Y.equal p q;;
Y.print p;;
```

polytest.ml

Exercise 18 (continued)

poly.ml

```
open Polynome

module I =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let plus = ( + )
    let mult = ( * )
    let equal = ( = )
    let print = print_int
end;;
module Xi = Make (I);;

let poly t =
  Array.fold_left
    (fun r (s, k) -> Xi.plus (Xi.monôme (int_of_string s) k) r)
    Xi.zéro
    (Array.mapi (fun i x -> (x, i)) t)
    ;;


let suffix t k = Array.sub t k (Array.length t - k);;

let main () =
  let écho, k =
    if Array.length Sys.argv > 1 &&  Sys.argv.(1) = "-v" then true, 2
    else  false, 1 in
  let t = suffix Sys.argv k in
  let p = poly t in
  let print_eval x =
    if écho then Printf.printf "P(%d) = " x;
    print_int (Xi.eval p x); print_newline () in
  if écho then (print_string "P = "; Xi.print p; print_newline ());
  while true do print_eval (int_of_string (read_line ())) done
    ;;

try main() with End_of_file -> ();;
```

poly.ml

Unix Shell

```
ocamlc -o poly polynome.mli polynome.ml poly.ml
./poly 1 2 1 << END
0
1
2
3
END
```

Unix Shell

Exercise 19, page 54

```
let rec fib n = if n > 1 then fib(n-1) + fb (n-2) else 1;;
```

Exercise 19 (continued)

```
let rec fib n = if n > 1 then fib(n-1) + fib (n-2) else 1;;

let _ =
```

```
  if Array.length Sys.argv > 1
  then
    try
      let n = int_of_string Sys.argv.(1) in
      print_int (fib n);
      print_newline()
    with Invalid_argument _ ->
      prerr_string "Expect one argument";
      prerr_newline();
      exit 1
  else
    begin
      prerr_string "Argument should be an integer";
      prerr_newline();
      exit 1
    end
        ;;
```

Exercise 20, page 54

```
type complex = { r : float; i : float };;
```

Exercise 20 (continued)

```
let ( ++ ) x y = { r = x.r +. y.r; i = x.i +. y.i }
and ( ** ) x y =
  { r = x.r *. y.r -. x.i *. y.i; i = x.r *. y.i +. x.i *. y.r };;
```

Exercise 20 (continued)

```
let unit = { r = 1.; i = 0. } and z0 = {r = 1.; i = 0.1}
let rec z n =  if n = 0 then unit else z (n-1) ** z0;;
z 99;;
```

Exercise 20 (continued)

```
let z' n = power ( ( ** ) z0 ) n unit;;
z' 99;;
```

Exercise 21, page 54

```
let rec length l =
  match l with
    h::t -> succ (length t)
  | [] -> 0;;
```

Exercise 21 (continued)

Non, car au retour, il faut ajouter un à la valeur rendue par l'appel récursif.

```
let length_fun l =
  let rec length n l =
    match l with
      h::t -> length (succ n) t
    | [] -> n in
  length 0 l;;
```

Exercise 21 (continued)

```
let rec find f l =
  match l with
    h::t -> if f h then h else find f t
 | [] -> raise Not_found;;
```

Exercise 22, page 55

```
let get s n = s.(n);;
let set s n c = s.(n) <- c;;
```

Exercise 22 (continued)

```
let init n f =
  if n = 0 then [||]
  else
    begin
      let a = Array.make n (f 0) in
      for i = 1 to n-1 do a.(i) <- f i done;
      a
    end;;
```

Exercise 22 (continued)

```
let make_matrix x y e =
 if x < 1 || y < 1 then raise (Invalid_argument "make_matrix")
 else init x (fun _ -> Array.make y e);;
```

Exercise 22 (continued)

```
let blit src src_pos dst dst_pos len =
  if
    src_pos + len > Array.length src ||
    dst_pos + len > Array.length dst ||
    src_pos < 0 ||
    dst_pos < 0
  then raise (Invalid_argument "blit")
  else for i = 0 to len-1 do dst.(dst_pos + i) <- src.(src_pos + i) done;;
```

Exercise 23, page 55

La version par défaut utilise l'égalité générique sur les clés. Il est parfois (voir souvent) nécessaire d'utiliser une fonction comparaison appropriée. On fabrique alors une nouvelle version des table de hash en passant sa propre fonction de comparaison dans une petite structure.

Exercise 23 (continued)

Une structure de donnée dont les clés contiennent un champ mutable. Il faut alors définir une fonctions de comparaison spécialisée qui ne prend pas en compte le champ mutable.