Programmation fonctionnelle avec OCaml

Nicolas Barnier, Cyril Allignol
nicolas.barnier@enac.fr

Objectifs

Objectifs

- Distinguer les **paradigmes** fondamentaux de la programmation impérative et de la **programmation fonctionnelle**
- Maîtriser les techniques de la programmation fonctionnelle pour produire des **programmes sûrs et concis**
- Maîtriser l'écriture des **fonctions récursives** et la transformation en fonction **récursive terminale**
- Savoir utiliser les **types algébriques** pour représenter les entités d'un problème ainsi que le **filtrage de motif** pour les traiter efficacement
- Programmer de manière **générique** en utilisant **ordre supérieur**, **polymorphisme** et **foncteurs**
- Maîtriser les **fichiers d'interface** des modules et l'**abstraction de type** pour écrire des applications sûres
- Savoir identifier les étapes de **compilation** et utiliser un **Makefile**

Principaux paradigmes de programmation

Langage	Impératif	Fonctionnel	Déclaratif	Objet
Exemples	Assembleur Fortran Pascal Basic C Ada	LISP (Scheme) **ML** (OCaml) Erlang F# (.NET)	Prolog C(L)P λProlog	C++ Eiffel Smalltalk Ada95 Java Python C# OCaml
Caract.	Mémoire (état) Séquence	Fonction Application	Relation Recherche	Héritage Messages
Modèle	Mach. de Turing	λ-calcul	Logique	—

Programmation impérative

Modèle

- Fondée sur le fonctionnement des **processeurs** (cf. machine de Turing, architecture Von Neumann)
- **Séquence** de modifications d'**état** : effets de bord (*side effects*)
- Opérations : séquence, affectation, boucle, branchement

Inconvénients

- Programmes souvent **plus difficiles à comprendre** (modifications de variables globales, code « spaghetti ») et à **corriger**
- **Peu expressif**, annotations de **type explicite** : programmes **plus longs et verbeux**
- **Gestion mémoire manuelle**, manipulation de pointeurs (C, C++)
- **Typage faible** pour de nombreux langages (C, C++, Python...)
- **Fuites** mémoires, **bugs** difficiles à éviter et à corriger

Intérêt de la programmation impérative

Efficacité et omniprésence

- Certains langages tendent à corriger ces défauts : Ada, Java, C#, Rust...
- Programmation **procédurale** : fonctions (sous-programmes), *possibilité* de programmer en style fonctionnel
- **Rapidité** : code proche de l'assembleur
- **Consommation mémoire faible**
- Certains algorithmes s'écrivent « naturellement » en style impératif
- Programmation **système**, accès **bas niveau** (C) : pilotes (*drivers*), code embarqué sur micro-contrôleur
- La plupart des langages de programmation et des programmes sont impératifs

Programmation fonctionnelle

Modèle

- Calcul = évaluation de fonctions imbriquées ($\neq$ modif mémoire)
- Opérations : abstraction (construction de fonctions), application
- λ-calcul typé [A. Church, 30s] : système formalisant la notion de **fonction calculable**, Turing-complet
- **Fondations théoriques sûres** : preuves d'absence d'erreur de type et de mémoire (voire de terminaison)
- Évaluation **stricte** (OCaml) ou **paresseuse** (Haskell)
- Langage **pur** sans référence (Haskell) ou **impur** (OCaml)

Intérêt de la programmation fonctionnelle

Productivité et sûreté

- Calcul d'**expressions**, **pas d'état** : plus simple, moins de bugs
- **Clarté** : découpage du code en petites fonctions (et modules) qui s'enchaînent, utilisation intensive des types (création, abstraction)
- **À réutiliser au maximum avec les autres langages !**
- **Expressivité** et **généricité** :
  - Syntaxe **concise** en adéquation avec la notion de fonction : composition, ordre supérieur, fonctions anonymes, application partielle
  - **Type algébrique** et **filtrage de motif** (ML $\neq$ Lisp)
  - **Polymorphisme paramétrique** et **foncteurs**
  - **Inférence de type** : pas d'indication de type
- **Productivité** : rapidité de **développement** d'algorithmes complexes, de **modification** de code et de **correction** d'erreur
- **Sûreté** : code embarqué (Lustre), certification de code (Coq, Tezos...)
- **Performances** : compilateur de code natif optimisé

OCaml

Multiparadigme : fonctionnel, impératif, objet

- Langage **fonctionnel** [Mc Carthy, 58]
- Dialecte **Caml** [INRIA, 85] de **ML** (*Meta Language*) [Milner, 78]
- Réimplémentation « légère » Caml Light [Leroy, 90]
- **Typage « fort »** : statique, inféré, polymorphisme paramétrique
- **Multi-paradigme** : fonctionnel, impératif, modules, objets
- **Gestion automatique de la mémoire** : *Garbage Collector*
- Compilateur de code natif optimisé très **efficace**
- Langage de référence pour l'initiation en France (avant Python...)

Implémentation

OCaml

- ocaml.org
- Équipe Cambium @ INRIA : cambium.inria.fr
- Multi-plateforme, multi-OS (mais développé sous Linux)
- Licence GPL
- Exécution :
  - **Interpréteur** (*top level*) : `ocaml`
  - **Compilateur** de **bytecode** (portabilité) : `ocamlc`
  - **Compilateur** de **code natif** (efficacité) : `ocamlopt`
- Gestionnaire de paquets `opam` : opam.ocaml.org
- Build system : dune.build

Autres implémentations de ML : Standard ML (SML/NJ, Moscow ML, MLton), Haskell, F# (.NET)...

Plan

Plan du cours

Bonjour monde !

C — hardi.c

```c
#include<stdio.h>
/* Block comment */
/* Global variable */
int a = 1729;

/* Function */
int f(int x) {
  int y = 2 * x;
  return y + a + 42;
}
/* main */
int main(){
  printf("%d\n", f(242));
  return 0;
}
```

OCaml — hardi.ml

```ocaml
(* Block comment *)
(* Global binding *)
let a = 1729

(* Function *)
let f = fun x ->
  let y = 2 * x in
  y + a + 42

(* main *)
let () =
  Printf.printf "%d\n" (f 242)
```


Valeurs, types et expressions

Valeur

- scalaire : booléen, entier, flottant, caractère...
- composée : tuple, enregistrement, chaîne...
- fonctionnelle

Toute valeur possède un type unique (et définitif)

- scalaire : `bool`, `int`, `float`, `char`...
- composé : `char * int`, `string`...
- fonctionnel : `float -> float -> float`

Expression

- constante, identificateur, arithmétique
- application de fonction : `atan2 (y2 - y1) (x2 - x1)`
- définition de fonction : `fun dy dx -> expr`
- liaison `let` : nommage

Nommage d'une expression par un identificateur

Liaison (*binding*) `let`

globale `let ident = expr`

locale
```
let f = fun x y ->
        ...
        let ident = expr in
        ...
```

- Cf. définition mathématique : "Let x be a positive integer..."
- Les liaisons **ne sont pas modifiables**
- Les liaisons peuvent être **masquées** (localement ou globalement mais pas détruites) par une autre liaison associée au même identificateur
- Les identificateurs doivent **commencer par une minuscule**

Typage

Inférence de type

- Définition des liaisons **sans annotation de type**
- Typage **statique** : le type des expressions est **inféré automatiquement** à la compilation

L'interpréteur affiche le type inféré et la valeur d'une expression

```
# 1729 mod 42;;
- : int = 7
# let pow = fun x y -> x ** y;;
val pow : float -> float -> float = <fun>
# let f = fun x -> x + 1;;
val f : int -> int = <fun>
# let f = (+) 1;;
val f : int -> int = <fun>
# f 2;;
- : int = 3
```

Type algébrique

Type produit (*product type*)

- Conjonction de types
- Tuple et enregistrement (*record*) :

```
# (42, "Doug");;
- : int * string = (42, "Doug")
```

Type somme (*variant type*)

- Disjonction de types, définie par l'utilisateur : `type id = type_expr`
- Énumération (sans paramètre, cf. C) :

```
# type bw = Black | White;;
type bw = Black | White
```

- Constructeurs avec paramètres :

```
# type roots = Zero | One of float | Two of float * float;;
type roots = Zero | One of float | Two of float * float
```

- Type récursif : `type seq = Empty | Elt of int * seq`

Traitement des types sommes

```
type answer = Yes | No | Unsure (* Enumeration *)

let invert = fun a ->
  match a with (* Simple pattern-matching *)
    Yes -> No
  | No -> Yes
  | Unsure -> Unsure

let definitive = fun a ->
  match a with
    Yes | No -> true (* Or pattern *)
  | Unsure -> false

let joint = fun a b ->
  match a, b with (* Pattern-matching over several values *)
    Yes, Yes | Yes, Unsure | Unsure, Yes -> Yes
  | No, No | No, Unsure | Unsure, No -> No
  | _ -> Unsure (* catch-all case *)
```

Filtrage de motif (*pattern-matching*)

```
match expression with
    pattern_1 -> expr_1
|   pattern_2 -> expr_2
...
|   pattern_n -> expr_n
```

- Les **motifs** sont **essayés dans l'ordre**
- Le **premier qui réussit est sélectionné** et l'expression correspondante est évaluée
- Le motif _ permet de reconnaître n'importe quelle expression
- Le compilateur **vérifie l'exhaustivité** du filtrage et indique les motifs non couverts → beaucoup de bugs détectés statiquement ! (éviter les cas « rattrape-tout »)

Belote

```
type couleur = Coeur | Carreau | Pique | Trefle

type carte =
  Petite of int * couleur
| Valet of couleur
| Dame of couleur
| Roi of couleur

let valeur = fun atout carte ->
 match carte with
   Petite (1, _) -> 11
 | Petite (9, c) -> if c = atout then 14 else 0
 | Petite (10, _) -> 10
 | Petite _ -> 0
 | Valet c -> if c = atout then 20 else 2
 | Dame _ -> 3
 | Roi _ -> 4
```

Représentation d'expression logique

```
type expr =
  False
| True
| Var of string
| Or of expr * expr
| And of expr * expr

let rec nb_nodes = fun e ->
  match e with
    False | True | Var _ -> 1
  | Or (e1, e2) | And (e1, e2) -> 1 + nb_nodes e1 + nb_nodes e2

let rec depth = ...

let rec nb_var = ...

let rec eval = ...
```


Portée des liaisons (*scope*)

Principe général : à définir au **plus proche** du contexte d'utilisation
$\longrightarrow$ plus simple à comprendre, éviter les utilisations erronées

Liaison globale — `let ident = expr`

- Située **en dehors d'une fonction** (ou d'une donnée)
- **Définie** (donc utilisable) pour toute **la suite** du fichier (voire pour d'autres fichiers)
- **Sauf** si elle est **masquée** par une liaison/paramètre de même nom

Liaison locale — `let ident = expr1 in expr2`

`expr1` est évaluée et sa valeur `ident` est utilisable dans `expr2`

```
let f = fun x y ->
  let u =
    if x > y then let x = x - y in x * x
    else x * y in
  x * x / u
```

Fonction

`fun param1 param2 ... -> expr`

- Tout est **expression** et « renvoie » une valeur : **pas** de `return` ni de `;`
- Définition et application : **pas** de parenthèse, virgule, type, accolade, double point, indentation significative...
- Nommage (global ou local) : `let f = fun arg1 arg2 -> ...`

Définition récursive

La récursivité est introduite par une liaison `let rec` :

```
let rec fact = fun n ->
  if n < 2 then 1 else n * fact (n - 1)
```

Appel de fonction

- Juxtaposition de la fonction et des arguments : `f (x+1) y (g z)`
- Prioritaire sur les opérateurs infixes : `f x + f y`
- Parenthésage : `f x y z` ≡ `(((f x) y) z)`

Type fonctionnel

- Les types fonctionnels sont désignés par des flèches `->` :

```
# atan2;;
- : float -> float -> float = <fun>
# String.init;;
- : int -> (int -> char) -> string = <fun>
```

- **Polymorphisme paramétrique** : variables de type `'a`, `'b`, `'c`...

```
# let max = fun x y ->
    if x >= y then x else y;;
val max : 'a -> 'a -> 'a = <fun>
# max 42 1729;;
- : int = 1729
# max "knuth" "karp";;
- : string = "knuth"
# let compose = fun f g x -> g (f x);;
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Fonction

Fonction anonyme

```
List.map (fun x -> x * x) l
```

Application partielle `fun x y ->` ≡ `fun x -> (fun y -> ...)`

```
# let successors = fun l -> List.map succ l;;
val successors : int list -> int list = <fun>
# let successors = List.map succ;;
val successors : int list -> int list = <fun>
# successors [1;2;3];;
- : int list = [2; 3; 4]
```


Plan du cours

Types prédéfinis

scalaire	type	valeurs	accès
unité	`unit`	`()`	
booléen	`bool`	`true`, `false`	
caractère	`char`	`'a'`, `'b'`...	
entier	`int`	1, -42...	
flottant	`float`	1.02, -2.7e52	
composé			
liste	`'a list`	`[1;2;3]`	`h :: t`
tuple	`'a * 'b *...`	`(1, 'a', 2.0)`	`(x, y, z)`
chaîne de caractères	`string`	"OCaml"	`s.[i]`
fonction	`int -> int -> int` `'a -> 'b -> 'a * 'b`	`fun x y -> x + y` `fun f s -> (f, s)`	

Typage

Inférence de type

- Le type **le plus général** est inféré :

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

- Les variables de types sont **substituées** par des types plus précis selon le contexte :

```
# List.map (fun x -> pow x 2.);;
- : float list -> float list = <fun>
```

- On peut renvoyer plusieurs valeurs à l'aide d'un tuple :

```
# let triplet = fun x y z -> (x, y, z);;
val triplet : 'a -> 'b -> 'c -> 'a * 'b * 'c = <fun>
```

- Les **types** inférés sont **fixés** par le compilateur et **ne peuvent jamais changer** (contrairement à Python) :
  e.g. les deux branches d'une conditionnelle doivent toujours renvoyer une valeur du même type (comme en C)

Type unit

- Types classiques : contient de l'information
- `unit` : type prédéfini avec une **unique** valeur `()`
- Renvoyée par les **effets de bord** :

```
# print_string "Hello World!\n";;
Hello World!
- : unit = ()
```

- Prise en paramètre des fonctions sans paramètre :

```
# Sys.time;;
- : unit -> float = <fun>
# Sys.time ();;
- : float = 0.052
```

- Analogue au `void` du C et au `None` de Python

Arithmétique

Opérateurs spécifiques pour les entiers et les flottants

- Les entiers et les flottants sont **incompatibles** :

```
# 1 + 2.5;;
Error: This expression has type float but an expression
       was expected of type int
# ( + );;
- : int -> int -> int = <fun>
# ( +. );;
- : float -> float -> float = <fun>
```

- **Pas de conversion automatique** mais des fonctions de conversion :

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

```
# truncate;;
- : float -> int = <fun>
```

- Les opérateurs ne sont **pas surchargés**
- Les opérateurs arithmétiques sur les flottants se terminent par le caractère '.' : `+.` `-.` `*.` `/.` sauf l'exponentiation : `**`

Comparaison

Opérateurs de comparaison (booléens) `= <> <= >= < >`

- Comparaison **polymorphe** : `'a -> 'a -> bool`
- On peut comparer **n'importe quel type** de valeur sauf les fonctions
- Comparaison **structurelle** :

```
# (1, "42") = (2 - 1, "4" ^ "2");;
- : bool = true
```

```
# [1; 2; 4] < [1; 0; 4];;
- : bool = false
```

- Attention : les opérateurs classiques == et != sont réservés pour l'**égalité physique** en mémoire (très rarement utilisés)

Fonction de comparaison : `val compare: 'a -> 'a -> int`

- Convient pour les fonctions de tri :

```
# List.sort compare [2;1;3;0];;
- : int list = [0; 1; 2; 3]
```

- `compare x y` renvoie un entier négatif si `x < y`, 0 si `x = y` et un entier positif si `x > y`

Arithmétique booléenne

Opérateur booléens `&& ||`

- **Conjonction** et **disjonction** :

```
# (&&);;
- : bool -> bool -> bool
```

```
# (||);;
- : bool -> bool -> bool
```

- Évaluation **paresseuse** (comme dans tous les langages)

Fonction de négation `not`

```
# not;;
- : bool -> bool = <fun>
```

Implication

```
# let (=>) = fun a b -> not a || b;;
val ( => ) : bool -> bool -> bool = <fun>
# true => false;;
- : bool = false
```

Conditionnelle

`if bool_expr then expr1 else expr2`

- Équivalent à la conditionnelle **expression** :

  Python `expr1 if cond else expr2`
  C `cond ? expr1 : expr2`

  ```
  if x mod 2 = 0 then x / 2 else 3*x+1
  ```

- **Toutes** les branches doivent renvoyer le **même type** :

  ```
  # if true then 1 else 2.4;;
  Error: This expression has type float but an expression
  was expected of type int
  ```

- Pas de syntaxe particulière pour les cascades :

  ```
  if cond1 then expr1
  else if cond2 then expr2
  ...
  else expr
  ```


Plan du cours

Exécution de programme

Différentes façons d'exécuter un programme

- Interprétation : évaluation directe du code source
  JavaScript/PHP, Shell, Lisp/Scheme, Matlab, Python, OCaml...
- Compilation : transformation du code source vers
  - du code assembleur pour un processeur donné (*code natif*)
    C, Fortran, Pascal, OCaml...
  - du code pour un pseudo-processeur, une machine virtuelle (*bytecode*)
    Java, Python, OCaml...
- Compilation à la volée (*Just In Time*)
  Java, PyPy (Python), C♯ (.NET)...

Le pour et le contre

	Interprétation	Compilation	
		Bytecode	Natif
Programme	`ocaml`	`ocamlc`	`ocamlopt`
Avantages	Simplicité	Développement	
	Portabilité		Efficacité
Inconvénients	Performance		Spécificité

Développement de programmes

1. **Écriture** du programme **source** à l'aide d'un **éditeur** doté de l'indentation automatique et de la mise en valeur de la syntaxe (couleurs) adaptées au langage (e.g. Emacs, mais pas gEdit)
2. Éventuellement : tests de *petites* fonctions dans l'interpréteur
3. **Compilation** (dans un terminal ou directement avec Emacs) : contrôle de propriétés **statiques**
   - Erreurs : doivent êtres corrigées
   - *Warnings* : doivent êtres corrigés aussi
4. **Exécution** : test de propriétés **dynamiques**
5. **Correction des erreurs** : il ne reste souvent plus que des erreurs de « haut niveau » dans la logique des algorithmes
   - exploitation de la trace des exceptions
   - affichages
   - ajout d'assertions (invariants vérifiés à l'exécution)
   - utilisation d'un *debugger* pas à pas

Exemple de programme exécutable : calcul de racines

```
type roots = Zero | One of float | Two of float * float

let roots = fun a b c ->
  let delta = b *. b -. 4. *. a *. c in
  if delta < 0. then Zero
  else
    let sr_delta = sqrt delta in
    let a2 = 2. *. a in
    if delta = 0. then One (-. b /. a2)
    else Two (-. (sr_delta +. b) /. a2, (sr_delta -. b) /.  a2)

let () = (* main *)
  let a = float_of_string Sys.argv.(1) in (* command-line *)
  let b = float_of_string Sys.argv.(2) in (* arguments *)
  let c = float_of_string Sys.argv.(3) in
  match roots a b c with
    Zero -> Printf.printf "No root\n"
  | One r -> Printf.printf "One root: %g\n" r
  | Two (r1, r2) -> Printf.printf "Two roots: %g and %g\n" r1 r2
```

Compilation et exécution du programme

Compilation

- `ocamlc -o roots roots.ml`
- `ocamlopt -o roots roots.ml`

Exécution (dans le même répertoire)

```
barnier@venar:~/ocaml$ ./roots 0.5 0.5 -3
Two roots: -3 and 2
```

Évaluation des liaisons

- Programme = séquence de liaisons `let` globales définissant des valeurs et des fonctions et **évaluées dans l'ordre** lors de l'exécution :

```
let radius = 5.
let surface = acos (-1.) *. radius          (* évaluée *)
let volume = fun height -> surface * height (* non évaluée *)
```

- **Une fonction n'est jamais évaluée si on ne l'applique pas à des arguments**

Exécution d'un programme

Initialisation du calcul

- Si le programme est **exécutable** ($\neq$ bibliothèque), on **initialise** le calcul principal avec une liaison sur `()`, cf. `main` en C :

```
let () =
  ...            (* command-line args and data processing *)
  let result = solve x y z in   (* call to main function *)
  Printf.printf ... result            (* result printing *)
```

- **Arguments de la ligne de commande** : tableau `Sys.argv`
  Exemple en lançant l'interpréteur avec la bibliothèque `Unix` :

```
barnier@venar:~/ocaml$ ocaml unix.cma
        OCaml version 4.05.0
# Sys.argv;;
- : string array = [|"/usr/bin/ocaml"; "unix.cma"|]
```

  Rq. : indexation du i^e élément d'un tableau `t` avec `t.(i)`

Modules

Structuration d'un programme en modules

- Chaque **fichier** est un **module**. La réciproque est fausse : un fichier peut contenir plusieurs sous-modules.
- La **première lettre** du nom du fichier est **capitalisée** : `mymodule.ml -> Mymodule`
- Une entité `x` dans un module `M` est désignée par `M.x`
- Rq. : les notations s'enchaînent pour un éventuel sous-module `M.Sub1.x`
- Exemples :

```
# String.length;;
- : string -> int = <fun>
# List.find;;
- : ('a -> bool) -> 'a list -> 'a = <fun>
```

- **Pas de dépendance mutuelle** entre modules

Module d'affichage

Affichage formaté — module `Printf`

- `printf format arg1 arg2` : sur la sortie standard
- `fprintf channel format arg1 arg2...` : écriture dans un fichier
- `sprintf format arg1 arg2...` : impression dans une chaîne
- `format` : chaîne contenant des **directives de formatage** ($\approx$ C)
  - un caractère ordinaire **autre que** % est affiché tel quel
  - %d entier en décimal
  - %c caractère
  - %s chaîne de caractères (string)
  - %f, %e, %g flottant
  - %a fonction d'affichage spécifique en argument supplémentaire
  - %! vide le buffer ($\equiv$ `flush channel`)
- Les autres arguments sont les valeurs correspondant aux directives **dans le même ordre**.

Bibliothèques de la distribution standard

Pervasives : core library (accessible directement)

- Types et exceptions de base : e.g. `type float`, `exception Exit`
- Arithmétique booléenne, entière et flottante, comparaisons, opérations d'entrée-sortie basiques

Standard

- **Structures de données** classiques : `List`, `Set`, `Map`, `Array`, `Queue`, `Hashtbl`...
- **Lecture** et **écriture formatées**, **système** : `String`, `Printf`, `Scanf`, `Sys`...

Autres

- Les *autres* bibliothèques de la distribution standard sont **dépendantes** de l'architecture : `Unix` (système), `Num` (nombres en précision arbitraire), `Graphics` (dessin)...
- Elles doivent être mentionnées dans la commande de compilation

Compilation et édition de liens

Étapes préalables

- Analyse **lexicale** : le source est découpé en **tokens** (constante, identificateur, opérateur, mot-clé et symbole)
- Analyse **syntaxique** : les constructions du langage sont reconnues (expression, liaison, fonction, structure de contrôle, types...)
- **Inférence** et vérification des **types**

Compilation et édition de liens

Production d'un exécutable

- La compilation produit du **code objet** `file.cmo` (ou `file.cmx` en code natif) à partir de `file.ml` (≡ `file.o` en C)
- Puis le code objet est **lié avec les bibliothèques** grâce au *linker* `ld` (directement invoqué par le compilateur) pour produire un exécutable
- Il est parfois nécessaire de mentionner explicitement une bibliothèque, fichier `.cma` (bytecode) ou `.cmxa` (code natif), à l'édition des liens :

```
# production de progsys.cmo
ocamlc -c progsys.ml
# édition des liens et production de l'exécutable
ocamlc -o progsys.out unix.cma progsys.cmo
# enchaînement automatique des deux étapes
ocamlc -o progsys.out unix.cma progsys.ml
```


Plan du cours

Récursivité

Répétition d'un traitement

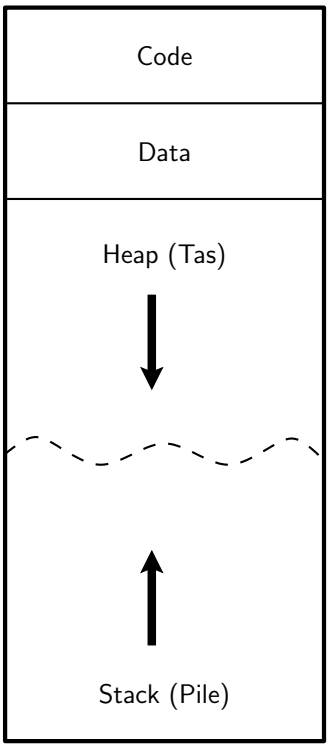
- Boucle bornée
- Boucle non bornée
- Appel fonctionnel : fonction récursive

Fonction récursive

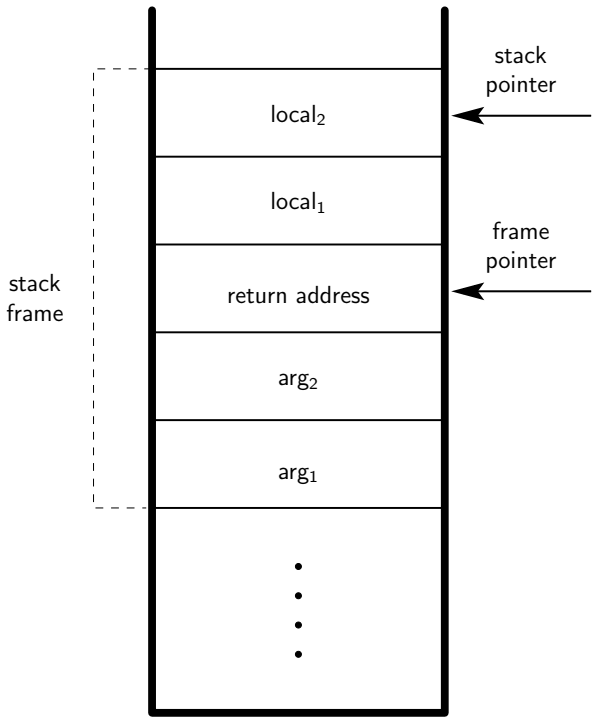
- Un appel à une fonction `f` est **récursif** s'il est situé dans le corps de la fonction `f`.
- La plupart des langages de programmation autorise les appels récursifs
- Une **pile** est **nécessaire** pour gérer :
  - les arguments des appels
  - les liaisons locales
  - l'adresse de retour
- Utilisation de mémoire **proportionnelle à la profondeur maximale** des appels récursifs (i.e. plus longue branche de l'arbre des appels)

Segmentation de la mémoire et pile d'exécution

Segments mémoire



Pile d'exécution (call stack)



Allocation dynamique

Tas

- L'**allocation dynamique** de la mémoire est **automatique**
- Cette mémoire est également **libérée automatiquement** : récupérateur de mémoire (*Garbage Collector*)
- En C, il faudrait un appel à `malloc` :

```
let f = fun n -> Array.make n 42
```

Clôture

OCaml permet également de construire des **clôtures** : fonction **locale** `f` renvoyée par la fonction englobante `g` et qui référence des **liaisons libres** (i.e. qui ne sont pas des paramètres de `f`) définies localement par `g` (et qui devraient donc être dépilées quand on sort de `g`...)

```
let addn = fun n ->
  let add = fun x -> n + x in      (* n est libre dans add *)
  add
```


Récupérateur de mémoire (*Garbage Collector*)

Récupération mémoire

1. **Arrêt** du programme quand toute la mémoire est consommée
2. **Marquage** de la mémoire *utile* : parcours des pointeurs
3. **Suppression** de la mémoire *inutile*
4. **Compactage** de ce qui reste (la mémoire utile)

GC incrémental générationnel

- Optimisé pour le rythme rapide d'allocation et libération de petites structures de données, typique des langages fonctionnels
- Les opérations les plus coûteuses ne sont pas exécutées à chaque cycle
- Le GC s'adapte au cycle de vie des objets (deux *générations*, deux algorithmes)
- L'algorithme peut s'interrompre et redémarrer dans le même état
- Paramétrable et contrôlable avec le module `Gc`

Récursion infinie

```
let rec main = fun () -> main ()
```

- Le `rec` permet d'augmenter la portée du `let` à la définition elle-même
- Pas de cas d'arrêt $\longrightarrow$ `Stack Overflow` : **dépassement** de pile
- Sauf si la fonction est **récursive terminale**
  $\longrightarrow$ optimisé en boucle par le compilateur (au moins `-O2` avec `gcc`)
- Sinon :

```
# let rec main = fun () ->
    let _ = main () in (* appel récursif non terminal *)
    ();;
val main : unit -> 'a = <fun>
# main ();;
Stack overflow during evaluation (looping recursion?).
```


Répétition contrôlée avec une conditionnelle

Fibonacci

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{cases}$$

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

Factorielle

$$\begin{cases} 0! &= 1 \\ n! &= n * (n-1)! \end{cases}$$

```
let rec fact = fun n ->
  if n = 0 then 1 else n * fact (n - 1)
```

Exécution

```
fact 3
if 3 = 0 then 1 else 3 * fact (3 - 1)
3 * (if 2 = 0 then 1 else 2 * fact (2 - 1))
3 * (2 * fact (2 - 1))
3 * (2 * fact 1)
3 * (2 * (if 1 = 0 then 1 else 1 * fact (1 - 1)))
3 * (2 * (1 * fact (1 - 1)))
3 * (2 * (1 * fact 0))
3 * (2 * (1 * (if 0 = 0 then 1 else 0 * fact (0 - 1))))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```


Preuve de terminaison

Règle

Une fonction récursive doit **toujours** posséder un cas **non** récursif.

Méthode

- Pour prouver qu'une fonction récursive termine, il faut trouver un ordre sur les arguments pour lequel les appels sont strictement décroissants :
  - raisonnement par récurrence
  - preuve par **induction** : sur les éléments d'un *ensemble défini par induction*, i.e. par un ensemble de base et des règles de production
  - nécessité d'un ordre *bien fondé* : pas de chaîne décroissante infinie
- C'est parfois impossible

  Ex. : suite de Syracuse $u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$

Style fonctionnel et récursivité

Comparaison avec les boucles

- Plus **puissant** que les boucles bornées (identique avec `while`)
- Ne nécessite **pas d'affectation** (pas de références)
- Raisonnement **simplifié** sur les programmes : **pas d'état** du calcul
- Consommation de mémoire **cachée** par l'usage de la pile
- La programmation fonctionnelle encourage le style récursif

Écrire une fonction récursive

- **Ne pas exécuter mentalement** les appels récursifs (trop complexe)
- **Supposer** que la fonction renvoie le résultat attendu **avant** de l'avoir écrite ≡ hypothèse de récurrence dans une démonstration par récurrence

Récursivité terminale

Calcul de racines carrées : méthode de Héron

On a $\lim_{n\to\infty} u_n = \sqrt{x}$ avec

$$\begin{cases} u_0 &= 1 \\ u_{n+1} &= \frac{1}{2}\left(u_n + \frac{x}{u_n}\right) \end{cases}$$

Version récursive directe

```
let rec u = fun x n ->
  if n = 0 then 1.
  else
    let un1 = u x (n-1) in
    (un1 +. x /. un1) /. 2.
```

Récursivité terminale

Inversion du sens des calculs

- Répétition avec une fonction récursive d'une transformation élémentaire $u_i \to u_{i+1}$:

$$u_i \to \frac{1}{2}\left(u_i + \frac{x}{u_i}\right)$$

- Où x reste constant mais est nécessaire parmi les paramètres :

$$(x, 1) \to \cdots \to (x, u_i) \to \left(x, \frac{1}{2}\left(u_i + \frac{x}{u_i}\right)\right)$$

- Ainsi que i et n pour savoir quand s'arrêter :

$$\cdots \to (x, n, i, u_i) \to \left(x, n, i+1, \frac{1}{2}\left(u_i + \frac{x}{u_i}\right)\right) \to \cdots \to (x, n, n, u_n)$$

Récursivité terminale

On renvoie l'accumulateur à la fin des calculs

```
let rec tailrec = fun x n i u ->
  if i = n then u
  else tailrec x n (i + 1) ((u +. x /. u) /. 2.)
```

Si i n'intervient pas dans la transformation ($\neq n!$), on peut initialiser le compteur i à n et décompter jusqu'à 0 :

```
let rec tailrec = fun x i u ->
  if i = 0 then u
  else tailrec x (i - 1) ((u +. x /. u) /. 2.)
```

Initialisation

- Mais la fonction nécessite plus de paramètres que la fonction initiale (condition initiale $u_0 = 1$, compteur i) : `tailrec x n 0 1.`
- Nécessité d'une fonction intermédiaire :

```
let racine = fun x n -> tailrec x n 1.
```

Récursivité terminale

Fonction locale

- Pour éviter une mauvaise initialisation, on utilise une **fonction locale** :

```
let racine = fun x n ->
  let rec racine_rec = fun x i u ->
    if i = 0 then u
    else racine_rec x (i - 1) ((u +. x /. u) /. 2.) in
  racine_rec x n 1.
```

- Les paramètres **constants** ne sont alors plus nécessaires :

```
let racine = fun x n ->
  let rec racine_rec = fun i u ->
    if i = 0 then u
    else racine_rec (i - 1) ((u +. x /. u) /. 2.) in
  racine_rec n 1.
```


Récursivité terminale

Critère d'arrêt

Pour le calcul d'une limite, la précision (ici au carré) peut être un meilleur critère :

```
let racine = fun x epsilon ->
  let rec racine_rec = fun u ->
    if abs_float (u *. u -. x) < epsilon then u
    else racine_rec ((u +. x /. u) /. 2.) in
  racine_rec 1.
```

Récursivité terminale

Factorielle

- La valeur de i est nécessaire au calcul :

$$(0,1) \to \cdots \to (i, f_i) \to (i+1, (i+1) \times f_i) \to \cdots \to (n, n!)$$

```
let fact = fun n ->
  let rec fact_rec = fun i fi ->
    if i = n then fi
    else fact_rec (i + 1) ((i + 1) * fi) in
  fact_rec 0 1
```

- Mais l'opération $(\times)$ est commutative, donc on peut quand même effectuer le calcul de manière descendante :

$$(n,1) \longrightarrow \cdots \longrightarrow (i, acc) \longrightarrow (i-1, i \times acc) \longrightarrow \cdots \longrightarrow (0, n!)$$

```
let fact = fun n ->
  let rec fact_rec = fun i acc ->
    if i = 0 then acc
    else fact_rec (i - 1) (i * acc) in
  fact_rec n 1
```


Exécution

```
fact 3
fact_rec 3 1
if 3 = 0 then 1 else fact_rec (3-1) (3*1)
fact_rec (3-1) (3*1)
fact_rec 2 3
if 2 = 0 then 3 else fact_rec (2-1) (2*3)
fact_rec (2-1) (2*3)
fact_rec 1 6
if 1 = 0 then 6 else fact_rec (1-1) (1*6)
fact_rec (1-1) (1*6)
fact_rec 0 6
if 0 = 0 then 6 else fact_rec (0-1) (0*6)
6
```

La taille de l'expression est **constante** et la consommation mémoire également.

Élimination des appels récursifs terminaux

Récursivité terminale standard

```
let f = fun a ->
  let rec f_rec = fun x y acc ->
    if cond then acc
    else f_rec expr_x expr_y expr_acc in
  f_rec x0 y0 acc0
```

Optimisée par les (bons) compilateurs en (pseudo-code)

```
x, y, acc := x0, y0, acc0;
tant que not cond
  x, y, acc := expr_x, expr_y, expr_acc
renvoyer acc
```


Recursivité mutuelle

Récursivité « cachée »

- `f` appelle `g` et `g` appelle `f`
- Définition `let rec` en **parallèle** avec le mot-clé `and` :

```
let rec est_pair = fun n ->
  n >= 0 && (n = 0 || est_impair (n-1))
and est_impair = fun n ->
  n >= 0 && (n = 1 || est_pair (n-1));;
```

Rq. : l'évaluation des opérateurs booléens est toujours *paresseuse*, i.e. l'opérande de droite n'est évalué que si nécessaire

Itérateurs

Soit f l'étape élémentaire d'une itération et z l'élément initial

- La répétition n fois s'exprime par la composition :

$$f(f(...f(z)...) = f^n(z)$$

- Une telle itération peut être générique : la fonction f et l'élément initial z deviennent les paramètres d'un itérateur général

Itérateur d'application

```
let rec iter = fun f n z ->
  if n = 0 then z else f (iter f (n-1) z)
let somme = fun x y -> iter succ y x
let produit = fun x y -> iter (somme x) y 0
let exp = fun x y -> iter (produit x) y 1
let knuth = fun x y -> iter (exp x) y 1
```


Itérateurs

OCaml est naturellement adapté au style fonctionnel

- Une fonction peut être **anonyme** :

```
let puissance_de_2 = fun n ->
  iter (fun x -> 2 * x) n 1
```

- Une fonction peut être appliquée **partiellement** :

```
let puissance_de_2 = fun n -> iter (( * ) 2) n 1
```

- L'itérateur est **polymorphe** :

```
val iter : ('a -> 'a) -> int -> 'a -> 'a = <fun>
let interets_a_10pourcent = fun n ->
  iter (fun x -> 1.10 *. x) n 1.
```

Itérateur *vs* fonction récursive

Avantages

- Concision d'écriture
- Preuve de **terminaison**

Version récursive terminale

```
let iter = fun f n z ->
  let rec iter_rec = fun i fi ->
    if i = n then fi
    else iter_rec (i+1) (f fi) in
  iter_rec 0 z;;
```


Récurseur

Système T de Gödel

- Si la transformation fait intervenir l'étape d'itération, on peut utiliser le *récurseur* suivant :

$$\begin{cases} F(0) &= z \\ F(n+1) &= f(n, F(n)) \end{cases}$$

```
let rec recurseur = fun f n z ->
  if n = 0 then z
  else f (n-1) (recurseur f (n-1) z)
```

- La factorielle est alors directement définie par :

```
let fact = fun n ->
  recurseur (fun x y -> (x+1) * y) n 1
```

Plan du cours

Le copier-coller est rarement profitable en programmation

Une expression recopiée *n* fois est *n* fois fausse

- Pas de constante numérique anonyme
- On finit toujours par modifier ce qu'on croyait fixé
- On fait toujours des erreurs
- Structurer le code rigoureusement dès le début

Le nommage permet d'éviter les répétitions

- Utilisation d'une liaison intermédiaire
- La complexité du calcul peut être affectée drastiquement
- Attention aux effets de bord !

Deux expressions semblables doivent être factorisées

- Fonction intermédiaire
- Passage en paramètre de ce qui diffère

Expressions identiques

Liaison intermédiaire

Pour éviter d'avoir à corriger une erreur à plusieurs endroits et mieux comprendre les traitements communs des différentes branches du code, l'expression suivante :

```
let f = fun x y ->
  if y > 0. then x*x + g y + 1
  else x*x + g y - 1
```

doit être « factorisée » avec une liaison intermédiaire :

```
let f = fun x y ->
  let x2gy = x*x + g y in
  if y > 0. then x2gy+1 else x2gy-1
```


Expressions identiques

Fonction intermédiaire

- **Globale** si elle est utile ailleurs :

```
let addsign = fun a b ->
  if a > 0. then b + 1 else b - 1
let f = fun x y -> addsign y (x*x + g y)
```

- Sinon **locale** :

```
let f = fun x y ->
  let addsign = fun a ->
    if y > 0. then a + 1 else a - 1 in
  addsign (x*x + g y)
```

- Voire **anonyme** (pas forcément très lisible...) :

```
let f = fun x y ->
  (fun a -> if y > 0. then a + 1 else a - 1)
  (x*x + g y)
```

Intermède sémantique

Sémantique de la liaison locale

Une liaison locale :

```
let x = e1 in e2
```

est équivalente à :

```
(fun x -> e2) e1
```

Donc une liaison peut être remplacée par une fonction à un paramètre :

```
let plus1 = fun x -> x + 1
let g = fun y ->
  let y1 = y+2 in plus1 (y1*y1)
```

pourrait s'écrire :

```
let g = fun y ->
  (fun f y1 -> f (y1*y1)) (fun x -> x+1) (y+2)
```

mais on n'y gagne pas toujours en clarté...

Expressions similaires

Deux fois la même expression à un renommage près

```
let h = fun (x1, y1) (x2, y2) ->
  (sqrt (x1 ** 2. +. y1 ** 2.), sqrt (x2 ** 2. +. y2 ** 2.))
```

Transformation avec une fonction à un paramètre

```
let norm = fun (x, y) -> sqrt (x ** 2. +. y ** 2.)
let h = fun p1 p2 -> (norm p1, norm p2)
```

Généralisation

Abstraction

- Nouvelle fonction ou fonction plus générale
- **Paramétrée par ce qui diffère** entre plusieurs expressions similaires

Intérêt

- **Concision**
- **Lisibilité** : le nom de la fonction **documente**
- **Maintenabilité** : une expression écrite 2 fois est 2 fois fausse
- Meilleure **compréhension** du code

Ordre supérieur

On peut abstraire par rapport à n'importe quoi (ou presque)

Un paramètre de la nouvelle fonction peut être une fonction

Aggrégation binaire sur un tableau

```
let product = fun t ->
  let n = Array.length t in
  let rec product_rec = fun i ->
    if i < n then t.(i) * product_rec (i + 1)
    else 1 in
  product_rec 0
let sum = fun t ->
  let n = Array.length t in
  let rec sum_rec = fun i ->
    if i < n then t.(i) + sum_rec (i + 1)
    else 0 in
  sum_rec 0
```

Ordre supérieur

On abstrait par rapport à la différence

On prend les valeurs et les opérations qui **diffèrent** en **paramètres** d'une fonction plus générale

```
let aggregate = fun init op t ->
  let n = Array.length t in
  let rec aggregate_rec = fun i ->
    if i < n then op t.(i) (aggregate_rec (i + 1))
    else init in
  aggregate_rec 0
let product = aggregate 1 ( * )
let sum = aggregate 0 ( + )
```

Rq. : la **fonction** correspondant à un opérateur **infixe** *op* se note : (*op*)
Donc 1+2*3 peut s'écrire : `( + ) 1 (( * ) 2 3)`

Plan du cours

Structures de données : types utilisateur

Types et structures de données

- Généralement, un module définit :
  - un type `t`
  - des valeurs particulières (e.g. ensemble vide)
  - des opérations de **créations**, d'**accès** et de **modification**
- Il faut pouvoir compléter les types de bases (types scalaires, chaînes, tableaux...) avec des **types nouveaux définis par le programmeur**
- À chaque nouvelle structure de données, on définira un nouveau type :

```
type ident = type_expr
```

- OCaml permet de définir des **types de données algébriques** et de traiter ces données par **filtrage de motif** (*pattern-matching*)

Type produit

Conjonction de types

- Représentation de **plusieurs valeurs simultanément**
- L'ensemble des valeurs est le **produit cartésien** des ensembles de chaque composante

Tuple

- Prédéfini
- Couples, triplets mais pas plus ! sinon : risque d'erreurs de position, modifications laborieuses

Enregistrement (*record*)

- À définir explicitement avec un nouveau nom
- Champs **nommés** : **documente** leur sens
- Notation plus adaptée au **développement incrémental**

Tuple

Tuple

- Il n'est pas nécessaire de définir le type au préalable
- Valeur : $(val_1, val_2, \ldots, val_n)$
- Type : $\tau_1 * \tau_2 * \cdots * \tau_n$

Exemples avec l'interpréteur (*toplevel*)

```
#(1, 2, 3);;
- : int * int * int = (1, 2, 3)

#let t = (2, "deux", '2');;
val t : int * string * char = (2, "deux", '2')

#let q = (1, "un", '1', t);;
val q : int * string * char * (int * string * char) =
  (1, "un", '1', (2, "deux", '2'))

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


Filtrage de motif (*pattern-matching*)

Accès aux composantes d'un tuple

On peut utiliser un **motif** (*pattern*) dans une définition `let` : c'est un motif de structure dont on nomme, avec des identificateurs, les éléments que l'on veut récupérer (ou _ pour une liaison anonyme)

```
#let (a, b, c) = t;;
val a : int = 2
val b : string = "deux"
val c : char = '2'

#let (_, _, _, d) = q;;
val d : int * string * char = (2, "deux", '2')
```

Filtrage de motif

Motifs

- Un motif peut être arbitrairement complexe :

```
#let f = fun x ->
#  let (_, ((_,a,_), b), _, _) = x in a + b
val f : 'a * (('b * int * 'c) * int) * 'd * 'e -> int = <fun>
```

- Les paramètres d'une fonction peuvent également être un motif :

```
#let f = fun (_, ((_,a,_), b), _, _) -> a + b
val f : 'a * (('b * int * 'c) * int) * 'd * 'e -> int = <fun>
```

Alias de motif (pattern as ident)

On peut nommer un motif pour éviter d'avoir à reconstruire des données :

```
let f = fun ((a, b) as ab) -> if a <= b then ab else (b, a)
let g = fun ((x, _) as xy) -> if x < 0 then (0, 0) else xy
```


Tuples *vs* tableaux

Caractéristiques des tuples

- Taille **statique** (ne peut pas dépendre d'un paramètre)
- **Non homogène**
- **Non modifiable**
- Accès **statique** aux éléments : pas d'index calculé (contrairement aux tuples de Python)

Enregistrement (*record*)

Inconvénient du produit cartésien

- Confusion possible entre champs de même type, par exemple pour un annuaire :

```
#("Jean", "10 rue Balzac", 31000, "Toulouse");;
- : string * string * int * string =
("Jean", "10 rue Balzac", 31000, "Toulouse")
```

- Impossible à utiliser quand le nombre de composantes devient trop important
- Il faut retoucher à tous les motifs si on ajoute une composante ultérieurement

Généralisation des tuples

- Chaque **champ** (*field*) est **nommé** par une étiquette (*label*)
- Le nouveau type résultant doit être **nommé** explicitement

Enregistrement

Définition — `type ident = type_expr`

```
#type client = {
#  nom : string;
#  rue : string;
#  cp : int;
#  ville : string
#};;
type client = {nom: string; rue: string; cp: int; ville: string;}
```

- `client` est un nouveau type
- l'ordre des champs n'est pas significatif
- `nom`, `rue`... sont les **étiquettes** (*labels*)
- une valeur pour ce type s'écrit :

```
#{rue = "du Lac"; ville = "Oo"; nom = "Beraldi"; cp = 31110};;
-: client = {nom="Beraldi";rue="du Lac";cp=31110;ville="Oo"}
```

Enregistrement

Un enregistrement ne peut pas être incomplet

```
# {nom = "Doe"};;
 Some record field labels are undefined: rue cp ville
```

Éviter d'utiliser la même étiquette dans deux types du même module

```
# type cl_num = {nom : string; numero : int};;
# let cp_nom = fun x -> (x.cp, x.nom);;
val cp_nom : client -> int * string = <fun>
# let nom_cp = fun x -> (x.nom, x.cp);;
 Error: The record type cl_num has no field cp
```

- La dernière étiquette définie masque la précédente...
- mais la première rencontrée dans l'expression donne le type
- Répartir les types dans différents modules :
  `x.Client.nom` du type `Client.t` *vs* `x.Cl_num.nom` du type `Cl_num.t`

Enregistrement : accès aux champs

Sélection

```
#let code_postal = fun i -> i.cp;;
val code_postal : client -> int = <fun>
```

Filtrage de motif (*pattern-matching*)

```
#let code_postal = fun {nom = _; rue = _; cp = code; ville = _} -
> code;;
 val code_postal : client -> int = <fun>
```

On peut ne faire apparaître que les champs nécessaires dans le motif :

```
#let code_postal = fun {cp = code} -> code;;
val code_postal : client -> int = <fun>
```

Type somme

Disjonction de types

Comment regrouper dans un même type des valeurs de types distincts ?

- Un type **somme** est une **union** finie et **étiquetée** de types
- Chaque étiquette de l'union est appelée **constructeur**
- Une constructeur est un identificateur **commençant par une majuscule**

Identification d'une personne par un nom **ou** son numéro de sécu

```
#type identification =
#    Nom of string
#  | Secu  of int;;
#let i1 = Nom "Doe";;
val i1 : identification = Nom "Doe"
#let i2 = Secu 1760173623128;;
val i2 : identification = Secu 1760173623128
```


Type algébrique

Disjonction de conjonctions (somme de produits)

Supposons que le nom soit accompagné de la date de naissance et que le numéro de sécu soit associé à une clé :

```
#type numero_secu = {num: int; cle:int};;
#type identification =
#    Nom of string * int
#  | Secu  of numero_secu;;
```

Les deux valeurs suivantes sont de même type :

```
#Nom ("Doe", 080176);;
- : identification = Nom ("Doe", 80176)
#Secu {num = 1760173623128; cle = 23};;
- : identification = Secu {num = 1760173623128; cle = 23}
```

Type algébrique

Figure géométrique

```
type point = {x: float; y: float}
type fig =
  Seg of point * point
| Rect of point * float * float
| Circle of point * float
let p = {x=8.; y=4.}
let c = Circle (p, 2.)
let r = Rect ({x=3.; y=3.}, 4., 2.)
let s = Seg ({x=5.; y=4.}, p)
let figs = [s; c; r]

val figs : fig list =
  [Seg ({x = 5.; y = 4.}, {x = 8.; y = 4.});
   Circle ({x = 8.; y = 4.}, 2.); Rect ({x = 3.; y = 3.}, 4., 2.)]
```


Reconnaissance des cas d'un type somme

Filtrage de motif (*pattern-matching*)

Généralisation du `switch` du C :

```
match expression with
   pattern_1 ->  result_1
|  pattern_2 -> result_2
...
|  pattern_n -> result_n
```

- Les **motifs** sont **essayés dans l'ordre**
- Le **premier qui réussit est sélectionné** et l'expression correspondante est évaluée
- Le compilateur **vérifie l'exhaustivité** du filtrage et indique les motifs non couverts → beaucoup de bugs détectés statiquement !

Filtrage de motif des types algébriques

```
let length = fun f ->
  match f with
    Seg (p1, p2) -> norm (sub p2 p1)
  | Rect (_, l, h) -> 2. *. (l +. h)
  | Circle (_, r) -> 2. *. pi *. r
val length : fig -> float = <fun>
let volume =  fun f ->
  match f with
    Seg _ -> 0.
  | Rect (_, l, h) -> l *. h
  | Circle (_, r) -> pi *. r ** 2.
let draw = fun f ->
  match f with (* coordinates should be ints for module Graphics *)
    Seg (p1, p2) ->
      Graphics.moveto p1.x p1.y; Graphics.lineto p2.x p2.y
  | Rect (p, l, h) -> Graphics.draw_rect p.x p.y l h
  | Circle (p, r) -> Graphics.draw_circle p.x p.y r
```


Généricité

Rappel : le copier-coller est rarement profitable en programmation

Exemple : structure de données dont le type des éléments est indifférent

```
#type paire_int = {e1 : int; e2 : int};;
#type paire_float = {r1 : float; r2 : float};;
#type paire_string = {c1 : string; c2 : string};;
```

Application d'une fonctions aux deux éléments d'une paire :

```
#let appl_int = fun f {e1=x; e2=y} -> f x y;;
#let appl_float = fun f {r1=x; r2=y} -> f x y;;
#let appl_string = fun f {c1=x; c2=y} -> f x y;;
```

Polymorphisme paramétrique

Paramètre de type

On abstrait les types `paire` par rapport au type de leurs éléments :

```
#type 'a paire = {e1 : 'a; e2 : 'a};;
```

On a défini un type **générique** paramétré.

```
#let appl = fun f {e1 = x; e2 = y} -> f x y;;
val appl : ('a -> 'a -> 'b) -> 'a paire -> 'b = <fun>
```

On a défini une fonction **polymorphe**.
On obtient des types différents pour différentes valeurs du paramètre `'a` :

```
# {e1 = 1; e2 = 2} = {e1 = 1.; e2 = 2.};;
 This expression has type float paire but is here used with
type int paire
```


Types récursifs

Récursivité structurelle

```
# type vqr = {taille: float; boucle_d_oreille: vqr};;
# {taille = 10.; boucle_d_oreille = {taille = 1.; boucle_d...
```

Un type récursif possède (au moins) un cas non récursif

Type algébrique récursif

```
type vqr =
    Infinitesimale
  | Boite of float * vqr

Boite (10., Boite (1., Boite (0.1, Infinitesimale)))
```

Traitement par filtrage

```
match expr with
  pattern1 -> expr1
| pattern2 -> expr2
  ...
```

```
#let rec profondeur = fun v ->
#  match v with
#    Infinitesimale -> 0
#  | Boite (_, b) -> 1 + profondeur b;;
val profondeur : vqr -> int = <fun>
```


Type algébrique récursif polymorphe

Liste : séquence d'éléments de même type

```
#type 'a liste =
#    Nil
#  | Cons of 'a * 'a liste;;
```

Arbre binaire générique avec des feuilles 'a et des nœuds 'b

```
#type ('a,'b) arbre2 =
#    Feuille of 'a
#  | Noeud of ('a,'b) noeud_binaire
#and ('a,'b) noeud_binaire = {
#    etiquette : 'b;
#    gauche:('a,'b) arbre2;
#    droit:('a,'b) arbre2
#};;
```

Type algébrique récursif polymorphe

Arbre quelconque générique avec des feuille 'a et des nœud 'b

```
#type ('a,'b) arbre =
#    F of 'a
#  | N of ('a,'b) noeud
#and ('a,'b) noeud = {
#  etiq : 'b;
#  fils : (('a,'b) arbre) liste};;
```

Arbre pour une expression boolénne

On préfèrera définir un nouveau type pour chaque utilisation :

```
#type eb =
#    False
#  | True
#  | Var of string
#  | Or of eb * eb
#  | And of eb * eb;;
```


Itérateurs

À chaque type correspond une fonction **intrinsèque** : son **itérateur**

Pour un type à n constructeurs

- l'itérateur est d'arité $n + 1$
- ses n premiers paramètres sont des fonctions d'arité égale aux arités des n constructeurs
- il est récursif si le type est récursif
- il possède n cas

L'itérateur permet de faire un **traitement uniforme** d'une donnée du type correspondant.

Récursivité structurelle

- Traitement récursif d'une structure de donnée récursive
- Preuve de terminaison par induction en utilisant l'itérateur

Itérateur pour les expressions booléennes

```
let eb_iter = fun c_f c_t f_var f_or f_and expr ->
  let rec iter = fun e ->
    match e with
      False -> c_f
    | True -> c_t
    | Var v -> f_var v
    | Or (e1, e2) -> f_or (iter e1) (iter e2)
    | And (e1, e2) -> f_and (iter e1) (iter e2) in
  iter expr

let nb_var = fun e ->
   eb_iter 0 0 (fun _ -> 1) ( + ) ( + ) e

nb_var (And (True, (Or (Var "a", Var "b"))));;
- : int = 2
```


Récapitulatif

En résumé

- Il est possible de définir des **nouveaux types**
- Les types peuvent être **paramétrés** par des **variables de type** : **polymorphisme paramétrique**
- Type **produit** : **produit** cartésien de types (tuples ou enregistrement)
- Type **somme** : **disjonction** de types
- Les types peuvent être **récursifs**
- À chaque type **algébrique** correspond un **itérateur** ■ intrinsèque ■

Plan du cours

Structure de donnée de LISP (LISt Processing)

Structure de séquence dynamique

Le type des listes (classique en programmation fonctionnelle) :

```
#type 'a list =
#    Nil
#  | Cons of 'a * 'a list;;
```

est prédéfini :

- `Nil` est noté `[]`
- `Cons`, noté `::`, est un constructeur **infixe**

Expression avec le constructeur à partir de la **tête** et de la **queue** :

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

Expression en extension :

```
#[[1] ; [1; 2]];;
- : int list list = [[1]; [1; 2]]
```

Expressions

```
#[(+) ; (-) ; (/)];;
- : (int -> int -> int) list = [<fun>; <fun>; <fun>]

#[1] = 1 :: [];;
- : bool = true

#let rec fact = fun n ->
#  if n = 0 then 1 else n * fact (n-1) in
#[fact 1; fact 2; fact 3; fact 4];;
- : int list = [1; 2; 6; 24]
```

Attention : la concaténation n'est pas gratuite !

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

Recopie de la première liste et **partage** de la seconde.

Parcours

Accès à la tête et à la queue

- Une liste est vide ou ne l'est pas !
- Le test doit se faire par **filtrage de motif**

Un motif peut utiliser les constructeurs :: et []

```
#let premier = fun l ->
#  match l with
#    [] -> None
#  | tete :: queue -> Some tete;;
val premier : 'a list -> 'a option = <fun>
```

Ou une expression en extension [x;y;...,z]

```
#let three = fun l ->
#  match l with
#    [_; _; _] -> true
#  | _ -> false;;
val three : 'a list -> bool = <fun>
```

Type récursif et fonction récursive

Traitement « naturel » des structures de données récursives

- Type récursif = Fonction récursive
- À un **constructeur récursif** correspond un **appel récursif**

```
#let rec longueur = fun l ->
#  match l with
#    [] -> 0
#  | tete :: queue -> 1 + longueur queue;;
val longueur : 'a list -> int = <fun>
```

```
#let rec somme = fun l ->
#  match l with
#    [] -> 0
#  | tete :: queue -> tete + somme queue;;
val somme : int list -> int = <fun>
```


Construction

Utilisation des constructeurs : : et []

Une liste se construit à partir de sa tête et de sa queue

Carré des éléments d'une liste (parcours et construction)

```
#let rec carres = fun l ->
#  match l with
#    [] -> []
#  | n :: ns -> n*n :: carres ns;;
val carres : int list -> int list = <fun>
```

Concaténation (@) : $[] \cup l_2 = l_2$ et $(a :: l_1) \cup l_2 = a :: (l_1 \cup l_2)$

```
#let rec conc = fun l1 l2 -> match l1 with
#    [] -> l2
#  | x :: xs -> x :: conc xs l2;;
val conc : 'a list -> 'a list -> 'a list = <fun>
```

Renversement

Renversement naïf (quadratique !) : $\overline{a :: l} = \bar{l} \cup [a]$

```
#let rec renverse = fun l ->
#  match l with
#    [] -> []
#  | x :: xs -> renverse xs @ [x];;
val renverse : 'a list -> 'a list = <fun>
```

Renversement récursif terminal (linéaire)

$$(l, []) \longrightarrow \cdots \longrightarrow (a :: l, r) \longrightarrow (l, a :: r) \longrightarrow \cdots \longrightarrow ([], \bar{l})$$

```
#let renverse = fun l ->
#  let rec rev = fun l r ->
#    match l with
#      [] -> r
#    | x :: xs -> rev xs (x :: r) in
#  rev l [];;
```


Itérateurs

Abstraction des traitements

Deux fonctions similaires sont des instances (cas particuliers) d'une troisième plus générale

```
#let rec carres = fun l ->
#  match l with
#    [] -> []
#  | n :: ns -> n*n :: carres ns;;
val carres : int list -> int list = <fun>

#let rec racines = fun l ->
#  match l with
#    [] -> []
#  | n :: ns -> sqrt n :: racines ns;;
val racines : float list -> float list = <fun>
```

Itérateur map

Abstraction de la fonction appliquée à chaque élément

Le traitement d'un élément est pris en paramètre de l'itérateur :

```
#let rec map = fun f l ->
#  match l with
#    [] -> []
#  | x :: xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#let carres = map (fun x -> x*x);;
val carres : int list -> int list = <fun>

#let racines = map sqrt;;
val racines : float list -> float list = <fun>
```

Cet itérateur est prédéfini :

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


Itérateur « intrinsèque » fold

Itérateur plus général

- L'itérateur map reconstruit systématiquement une liste de même taille
- L'itérateur `fold f [x1;x2;...;xn] z` renvoie la valeur :

  `f x1 (f x2 ...(f xn z)...)`

  où `f` est une fonction à deux paramètres et `z` une valeur

```
#let rec fold = fun f l z ->
#  match l with
#    [] -> z
#  | x :: xs -> f x (fold f xs z);;
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

#let somme = fun l -> fold ( + ) l 0;;
val somme : int list -> int = <fun>
#somme [1;2;3];;
- : int = 6
#List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

fold_left

Version récursive terminale

- **Inverse** la liste si on la reconstruit
- `fold_left f acc [x1;x2;...;xn]` renvoie
  `f (...(f (f acc x1) x2) ...) xn`

```
#let rec fold_left f acc l =
#  match l with
#    [] -> acc
#  | a :: l -> fold_left f (f acc a) l;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

#List.fold_left ( * ) 1 [1;2;3];;
- : int = 6

#List.fold_left (fun acc x -> x :: acc) [] [1;2;3];;
- : int list = [3; 2; 1]
```

Rq. : Ni `Array.fold_left` **ni** `Array.fold_right` ne sont récursives

Liste d'association

Utilisation comme dictionnaire

La liste de couples est une implémentation simple d'un ensemble d'associations clé $\rightarrow$ valeur

Annuaire nom $\rightarrow$ numéro

```
#let annuaire = [("A", 4053); ("B", 4142); ("C", 4282)];;
```

Recherche de la valeur associée à une clé :

```
#let rec assoc = fun k l ->
#  match l with
#    [] -> raise Not_found
#  | (k', v) :: kvs -> if k = k' then v else assoc k kvs;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>

#assoc "B" annuaire;;
- : int = 4142
```

La fonction `raise` abandonne le calcul et *lève* une exception

Liste d'association

Garde associée à un motif — `pattern when condition`

- On peut ajouter une **condition** quelconque à un motif
- Elle peut porter sur les identifiants liés par le motif

```
#let rec assoc = fun k l ->
#    match l with
#    [] -> raise Not_found
#  | (k', v) :: kvs when k = k' -> v
#  | _ :: kvs -> assoc k kvs;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

Prédéfini :

```
#List.assoc;;
- : 'a -> ('a * 'b) list -> 'b = <fun>
```


Plan du cours

Modularité

Un programme sera *structuré* en

- **fichiers** : unité de compilation
- **modules** : chaque fichier est un module pouvant contenir
  - des types, des exceptions
  - des valeurs, des fonctions
  - des sous-modules
- **bibliothèques** : collection de fichiers (donc de modules)

Avantages

- Taille « raisonnable » pour chaque fichier
- **Espace de nommage** (*namespace*)
- **Restriction d'interface**
- **Compilation séparée**

Compilation et édition de liens

Compilation

Production d'un **objet** (`.cmo` avec `ocamlc`, `.cmx` avec `ocamlopt`) à partir du **source** avec l'option `-c` du compilateur :
`ocamlc -c file.ml` compile `file.ml` et produit `file.cmo`

Édition de liens

Production d'un **exécutable** en **liant** un ou plusieurs *objets* et des bibliothèques :
`ocamlc -o file file.cmo` lie `file.cmo` à la bibliothèque standard et produit l'exécutable `file`
L'édition de liens fait la correspondance entre les appels aux fonctions des bibliothèques et le code de ces fonctions

Compilation séparée de plusieurs fichiers

Code source réparti dans plusieurs fichiers

- Compilation :

```
ocamlc -c file1.ml
ocamlc -c file2.ml
ocamlc -c file3.ml
```

- Édition de liens :

```
ocamlc -o prog file1.cmo file2.cmo file3.cmo
```


Édition de liens avec plusieurs fichiers

Liens entre unités de compilation

Chaque unité de compilation constitue un **espace de nommage** distinct : le nom `x` dans `file1.ml` (module `File1`) est différent du nom `x` dans `file2.ml` (`File2`)

Espace de nommage

- Le nom `x` défini (avec une liaison **globale**) dans le fichier `file1.ml` est désigné `File1.x` partout en dehors de `file1.ml`
- La directive `open File1` donne la **visibilité** sur tous les noms définis dans `file1.ml`. **Non recommandé :** la notation pointée **documente** (`Array.length`, `List.length`...)

Gestion des dépendances

$\neq$ #include en C

Soit `file1.ml` définissant `f` et `file2.ml` utilisant `File1.f`

- Le fichier `file1.ml` doit être compilé (avec l'option `-c`) **avant** `file2.ml`
- Lors de la compilation de `file2.ml`, le compilateur **vérifie** que le nom `f` est bien défini dans `File1`
- Lors de l'édition de liens de `file2.cmo`, `file1.cmo` doit être présent et placé **avant** dans la commande de compilation :

  `ocamlc -o executable ... file1.cmo ... file2.cmo ...`
- Conséquence : **pas de dépendances croisées** possibles entre unités de compilation

Fonctionnement : lors de la compilation de `file1.ml`, `ocaml` produit `file1.cmi` qui contient la description (i.e. le type) de tous les noms définis dans `File1`

Fichiers d'interface en **OCaml**

Restrictions sur les noms exportés

- Noms globaux visibles par défaut dans tout autre fichier.
- **Restriction de la visibilité** en définissant une **interface** `file1.mli` des valeurs exportées par l'**implémentation** `file1.ml`

file1.ml

```
type t = {x: float; y: float}
let ma_valeur_a_moi = 7
let pi = acos (-1.)
let ma_fonction_a_moi = fun x y -> x ** (1. /. y)
let fonction_utile = fun k pt -> {x = k *. pt.x; y = k *. pt.y}
```

file1.mli

```
type t = {x: float; y: float}
val pi : float
val fonction_utile : float -> t -> t
```

Compilation des interfaces

Compilation

1. `ocamlc file1.mli` produit `file1.cmi` (interface compilée)
2. Compilation de tout ce qui dépend de `File1`, `file1.ml` y compris, avec **vérification** de la cohérence (existence et type)
3. Édition de liens

Génération de l'interface

L'option `-i` du compilateur permet de générer une interface par défaut où **tout est exporté** :

```
$ ocamlc -i file1.ml
type t =  x : float; y : float;
val ma_valeur_a_moi : int
val pi : float
val ma_fonction_a_moi : float -> float -> float
val fonction_utile : float -> t -> t
```

Redirection de la sortie standard puis édition du `.mli` :

```
$ ocamlc -i file1.ml > file1.mli
```


Abstraction de type

L'**implémentation d'un type** peut être **cachée** par l'interface du module

stak.ml

```
type 'a t = {stack : 'a list; size : int}
exception Empty
let empty = {stack = []; size = 0}
let pop = fun s ->
  match s.stack with
    [] -> raise Empty
  | h :: t -> (h, {stack = t; size = s.size - 1})
```

stak.mli

```
type 'a t
(** The type of stacks containing elements of type ['a]. *)
exception Empty
val empty : 'a t
(** Empty stack. *)
val pop : 'a t -> 'a * 'a t
(** [pop s] returns the top element and the rest of stack [s].
    Raises exception [Empty] if [s] is empty. *)
```

Abstraction de type

Type de données **abstrait** (opaque)

```
# let queue = Stak.empty;;
val queue : 'a Stak.t = <abstr>
# let queue = Stak.push 4104 queue;;
val queue : int Stak.t = <abstr>
# let list = queue.Stak.stack;;
Error: Unbound record field Stak.stack
# let (top, queue) = Stak.pop queue;;
val top : int = 4104
val queue : int Stak.t = <abstr>
```

Nécessité d'écrire des fonction d'accès

- Cf. méthodes *getter* et *setter* en programmation objet
- Exporter le minimum, i.e. uniquement ce qui est :
  - requis : nécessaire pour utiliser la structure de données
  - sûr : préserver l'intégrité des données

Intérêts

Sûreté : préserver les **invariants** (axiomes) de la structure

Manipulation de la donnée par l'intermédiaire de fonctions qui le garantissent

Fonction `Stak.push`

`let push = fun x s -> ...`
Si le type est explicite, on peut violer les invariants :
`let s = {Stak.stack = []; size = 1}`
Les fonctions qui reposent dessus deviennent incorrects

```
let pop = fun s ->
  if s.size = 0 then raise Empty else ...
```

Développement : transparence de l'optimisation de code

Différentes implémentations pour la même interface **sans changer le code** qui l'utilise

(Re-)Compilation automatique et efficace

- Les commandes de compilations peuvent être longues et fastidieuses
- Pour un programme constitué de **plusieurs fichiers**, il n'est pas nécessaire de systématiquement **tout** recompiler après une modification, mais seulement ce qui **dépend** des modifications

Build system

- (re-)compilation du strict nécessaire avec une seule commande
- plusieurs exécutables (ou bibliothèques) possibles
- fichier(s) de paramétrage
- génération de documentation, tests, installation, distribution...
- `make` : générique, omniprésent, historique
- `dune` : spécifique à OCaml, gestion des bibliothèques, simplicité

`make`

Générique, omniprésent, historique

- On saisira dans un fichier nommé (par convention) `Makefile` la description des opérations nécessaires pour la compilation
- La recompilation sera **exécuté** avec la commande `make` (éventuellement suivi d'un nom de *cible*)
- Un `Makefile` est consitué de variables et de **règles** cible/dépendances/commande
- L'outil `make` ne **recompile que ce qui est nécessaire**
- Il peut aussi servir à générer la documentation, installer un programme, produire des paquets, nettoyer le répertoire de travail...

Makefile ad-hoc

Makefile

```
# Makefile (les commentaires commencent par un croisillon)
tp : tp1.cmo tp2.cmo
        ocamlc -o tp tp1.cmo tp2.cmo

tp1.cmi : tp1.mli
        ocamlc tp1.mli

tp1.cmo : tp1.cmi tp1.ml
        ocamlc -c tp1.ml

tp2.cmo : tp2.ml
        ocamlc -c tp2.ml
```

La recompilation sera déclenchée en exécutant simplement la commande `make` dans le répertoire où sont situés les fichiers sources et le `Makefile`

Règles de production

Règles

Chaque **règle** de la forme :

cible : *dependances*
<tabulation>*commande*

- exprime que les **dépendances** sont nécessaires pour fabriquer la **cible**
- si la cible est **plus ancienne** (comparaison des dates des fichiers) que l'une des dépendances, alors la **commande** est exécutée
- si l'une des dépendances n'existe pas encore mais qu'il y a une règle pour la fabriquer, elle sera déclenchée

Rq. : on peut utiliser `make` pour générer n'importe quel type de projet (e.g. ce cours a été produit avec un `Makefile` qui utilise `pdflatex`, `fig2dev`, `pdfnup`...)

Règles génériques

Règles génériques

- Pour éviter de dupliquer des règles similaires où seul le *nom de base* du fichier change :

```
%.cmi: %.mli
        ocamlc $<
```

- % désigne n'importe quel nom de base de fichier
- $< désigne le nom de la première dépendance

Variables

- On peut également définir des **variables** dans un `Makefile` :

```
SOURCES = tp1.ml tp2.ml
OBJS = $(SOURCES:.ml=.cmo)
prog: $(OBJS)
        ocamlc -o $@ $^
```

- $^ désigne *toutes* les dépendances
- $@ désigne le nom de la cible

Makefile générique

```
SOURCES = file1.ml file2.ml
TARGET = prog
OCAMLC = ocamlc -g
DEP = ocamldep
OBJS = $(SOURCES:.ml=.cmo)

all: .depend byte

byte: $(TARGET)

$(TARGET): $(OBJS)
        $(OCAMLC) -o $@ $^

%.cmi: %.mli
        $(OCAMLC) $<

%.cmo: %.ml
        $(OCAMLC) -c $<
```

```
.PHONY: clean

clean:
        rm -f *.cm[io] *~

.depend: $(SOURCES)
        $(DEP) *.mli *.ml > .depend

include .depend
```

Dune : « a build system for OCaml »

- Aucune configuration nécessaire pour un projet simple
- Commandes : `dune {init|build|test|exec|install|clean}`
- Initialisation de la structure des répertoires du projet

```
dune init proj myproj
```

```
myproj/
   dune-project          -> description
   myproj.opam
   bin/                  -> code source exécutables (.ml)
      dune               -> configuration
      main.ml
   _build/               -> fichiers générés (dont exécutables)
   lib/                  -> code source fonctionnalités (.ml, .mli)
      dune
   test/                 -> scripts de test
      dune
      myproj.ml
```


Structure, compilation et exécution

- `lib` : modules .ml et interfaces .mli
- `bin` : code source de l'exécutable, qui peut utiliser `Myproj.Mod` si `mod.ml` (et `mod.mli`) présents dans `lib`
- un fichier `dune` par répertoire de code source (`bin`, `lib` et `test`)
- (re-)compilation automatique avec la commande `build` (depuis un sous-répertoire quelconque du projet) :

  ```
  cd myproj
  dune build
  ```

  tous les fichiers générés, dont les exécutables, sont placés dans `_build/default`
- exécution avec la commande `exec` dans le répertoire racine :

  ```
  dune exec myproj
  ```

Fichiers de configuration dune

bin/dune

```
(executable
 (public_name myproj)
 (name main)
 (libraries myproj))
```

Ajout de bibliothèques : `(libraries myproj unix)`

lib/dune

```
(library
 (name myproj))
```

On peut ajouter des bibliothèques de la même manière

Documentation : dune.readthedocs.io

Plan du cours

Foncteur (*functor*)

Fonction des modules vers les modules

- À ne pas confondre avec les *function objects* (parfois appelés *functors*) de certains langages (e.g. C++)
- Similaires aux *templates* (C++) et *paquets génériques* (Ada)
- « Les foncteurs sont aux modules ce que les fonctions sont aux valeurs »
- Correspondent aux foncteurs de la *théorie des catégories*

Généricité de masse

- Évite d'avoir à passer de nombreux paramètres aux fonctions génériques
- Utilisés dans la bibliothèque standard pour les structures de données ordonnées `Set`, `Map` et les tables de hachage `Hashtbl`

Définition de module

Structure

- **Différent** des *structures* en C (appelées *enregistrements* en OCaml : e.g. `type point = {x: float; y: float}`)
- Les **structures** en OCaml désignent des collections de définitions de valeurs, types, exceptions, (sous-)modules...
- La définition commence par `struct` et se termine par `end`

Syntaxe

```
struct
  type t = ...
  exception E of t * string
  let zero = ...
  let f = fun x y -> ...
  module Sub = struct ... end
  ...
end
```

Nommage de module

Liaison module

```
module M = struct ... end
```

Point 2D

```
module Point = struct
  type t = {x: float; y: float}
  let null = {x = 0.; y = 0.}
  let add = fun p q ->
    {x = p.x +. q.x; y = p.y +. q.y}
  let mul = fun k p ->
    {x = k *. p.x; y = k *. p.y}
  let scalprod = fun p q ->
    p.x *. q.x +. p.y *. q.y
  let norm = fun p ->
    sqrt (p.x *. p.x +. p.y *. p.y)
end
```


Signature de module

Type des modules

- Le type d'un module est appelée une **signature**
- Une signature est une collection de spécifications de type, comme celle spécifiée par un fichier d'interface `.mli`
- Une signature commence par `sig` et se termine par `end`

Syntaxe

```
sig
  type t [= ...]
  exception E of t * string
  val zero : t
  val f : t -> t -> ...
  module Sub : S
  ...
end
```

Inférence de signature

Inférence de type et modules

Par défaut, l'inférence de type synthétise la signature

- pour **toutes les valeurs** d'un module
- la **plus générale** possible

```
# module P = struct
    type t = float * float
    let add = fun (x1, y1) (x2, y2) -> (x1 +. x2, y1 +. y2)
    let add1 = add (1., 1.)
  end;;
module P :
  sig
    type t = float * float
    val add : float * float -> float * float -> float * float
    val add1 : float * float -> float * float
  end
```


Restriction et abstraction par une signature

`module M : SIG = struct ... end`

Comme avec les fichiers d'interface qui spécifient la signature d'un module définie dans une unité de compilation (i.e. un fichier), on peut avec une signature :

- **restreindre** les valeurs
- **abstraire** les types

exportés par un module

```
# module P : sig type t val add : t -> t -> t end = struct
    [...]
  end;;
module P : sig type t val add : t -> t -> t end
```

Nommage de signature

Liaison module type

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

```
# module type S = sig
    type t
    val make : float -> float -> t
    val add : t -> t -> t
  end;;
module type S = sig type t val add : t -> t -> t end
# module P : S = struct
    type t = float * float
    let make = fun x y -> (x, y)
    let add = fun (x1, y1) (x2, y2) -> (x1 +. x2, y1 +. y2)
    let add1 = add (1., 1.)
  end;;
module P : S
# P.add1;;
Error: Unbound value P.add1
```


Définition de foncteur

Module paramétré

- Un **foncteur** (*functor*) est un module qui prend en paramètre un autre module `M` et définit ses valeurs (types, exceptions...) en utilisant les entités exportées par `M`
- La **signature** du paramètre doit être indiquée **explicitement**

  ```
  functor (M : SIG) -> struct ... end
  ```

- **Nommage** : les foncteurs définis dans un fichier (module) `f.ml` sont souvent nommés `Make`

  ```
  module Make = functor (M : SIG) -> struct ... end
  ```

- Pour pouvoir utiliser les valeurs définies dans un foncteur, il faut l'**appliquer** à un module qui **implémente** la signature `SIG`

  ```
  module MF = F.Make(M)
  ```

- Un foncteur peut prendre **plusieurs modules** en paramètres
- Les différentes applications du foncteur **partagent leur code**

Application de foncteur

Foncteur défini dans bTree.ml (module BTree)

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

module Make = functor (Ord : OrdType) -> struct
  type elt = Ord.t
  type t = Empty | Node of t * elt * t
  exception EmptyTree
  let rec mem = fun x t ->
    match t with
      Empty -> raise EmptyTree
    | Node (l, y, r) ->
        let c = Ord.compare x y in
        c = 0 || mem x (if c < 0 then l else r)
end
```


Application de foncteur

Application à des entiers

```
# module Int = struct
    type t = int
    let compare = compare
  end;;
# module IntBTree = BTree.Make(Int);;
module IntBTree :
  sig
    type elt = Int.t
    type t = BTree.Make(Int).t = Empty | Node of t * elt * t
    exception EmptyTree
    val mem : Int.t -> t -> bool
  end
```

Application de foncteur

Application à des vecteurs

```
# module Vect = struct
    type t = Point.t
    let compare = fun u v ->
      compare (Point.norm u) (Point.norm v)
  end;;
# module VectBTree = BTree.Make(Vect);;
module VectBTree :
  sig
    type elt = Vect.t
    type t = BTree.Make(Vect).t = Empty | Node of t * elt * t
    exception EmptyTree
    val mem : Vect.t -> t -> bool
  end
```


Abstraction de type dans les foncteurs

- Il est préférable d'**abstraire** la représentation des arbres (pour pouvoir en changer sans « casser » le code des utilisateurs)
- Définition d'une **signature** pour restreindre le **résultat de l'application du foncteur**
- Plus d'accès au constructeur de type algébrique : nécessité d'exporter des « constructeurs » pour pouvoir créer des valeurs de type `t`

Signature du résultat du foncteur — `bTree.ml` et `bTree.mli`

```
module type S = sig
  type elt
  type t
  exception EmptyTree
  val empty : t
  val add : elt -> t -> t
  val mem : elt -> t -> bool
end
```

Abstraction de type dans les foncteurs

Restriction de la signature — bTree.mli

```
module Make : functor (Ord : OrdType) -> S
```

Tentative d'utilisation du foncteur...

```
# module IntBTree = BTree.Make(Int);;
module IntBTree :
  sig
    type elt = BTree.Make(Int).elt
    type t = BTree.Make(Int).t
    exception EmptyTree
    val empty : t
    val mem : elt -> t -> bool
    val add : elt -> t -> t
  end
# IntBTree.add 12 IntBTree.empty;;
Error: This expression has type int but an
expression was expected of type IntBTree.elt =
BTree.Make(Int).elt
```


Abstraction de type dans les foncteurs

Contraintes de type

- Le type des éléments de l'arbre est maintenant abstrait et incompatible avec celui du module `Int` (`type t = int`)
- Il faut ajouter une **contrainte de type** à la signature pour spécifier des **équations de type** :
  `S with type t1 = type-expr1 and type t2 = type-expr2 [and...]`
  où `t1` (`t2`...) sont des types spécifiés dans la signature `S` et `type-expr1` (`type-expr2`...) peuvent utiliser les types exportés par le(s) paramètre(s) du foncteur

Signature avec abstraction et contrainte de type — bTree.mli

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

Abstraction de type dans les foncteurs

Utilisation du résultat du foncteur abstrait

```
# module IntBTree = BTree.Make(Int);;
module IntBTree :
  sig
    type elt = Int.t
    type t = BTree.Make(Int).t
    exception EmptyTree
    val empty : t
    val mem : elt -> t -> bool
    val add : elt -> t -> t
  end
# IntBTree.add 12 IntBTree.empty;;
- : IntBTree.t = <abstr>
```


Sûreté du typage

Incompatibilité des différentes applications du foncteur

```
# module AbsInt = struct
    type t = int
    let compare = fun x y -> compare (abs x) (abs y)
end  ;;
module AbsInt : sig type t = int val compare : int -> int -> int end
# module AbsIntBTree = BTree.Make(AbsInt);;
module AbsIntBTree :
  sig
    type elt = AbsInt.t
    type t = BTree.Make(AbsInt).t
    [...]
  end
# IntBTree.add 42 AbsIntBTree.empty;;
Error: This expression has type
       AbsIntBTree.t = BTree.Make(AbsInt).t
       but an expression was expected of type
       IntBTree.t = BTree.Make(Int).t
```

Foncteurs de la bibliothèque standard

Modules Set et Map

- Ces modules implémentent respectivement les types de données « ensemble ordonné » et « table d'association » (*dictionnaire*)
- Ils prennent en paramètre un module d'élément ordonné de signature :

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

- Module Set :

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

- Module Map :

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


Foncteurs de la bibliothèque standard

Signature Set.S

```
type elt
type t
val empty : t
val mem : elt -> t -> bool
[...]
```

Signature Map.S

```
type key
type 'a t
val empty : 'a t
val mem : key -> 'a t -> bool
[...]
```

Application

```
# module IntSet = Set.Make(Int);;
module IntSet :
 sig
  type elt = Int.t
  type t = Set.Make(Int).t
  val empty : t
  val mem : elt -> t -> bool
  [...]
 end
```

Application

```
# module VectMap = Map.Make(Vect);;
module VectMap :
 sig
  type key = Vect.t
  type 'a t = 'a Map.Make(Vect).t
  val empty : 'a t
  val mem : key -> 'a t -> bool
  [...]
 end
```

Foncteurs de la bibliothèque standard

Module `Hashtbl`

- Implémente le type de données « table d'association »
- **Table de hachage** :
  - modification **en place**
  - clés **non-ordonnées**
  - accès en **temps constant**
- Création avec une **taille initiale** puis agrandissement pour maintenir le **facteur de charge**
- La dernière association clé/valeur ajoutée **masque** la précédente si les clés sont égales (mais ne la détruit pas $\neq$ Python) : `findall` renvoie la liste de toutes les éléments associés à une clé
- Interface **polymorphe** (utilise `hash : 'a -> int`) :

```
type ('a, 'b) t
val create : int -> ('a, 'b) t
val add : ('a, 'b) t -> 'a -> 'b -> unit
[...]
```


Foncteurs de la bibliothèque standard

Interface fonctorielle du module `Hashtbl`

- Le foncteur `Make` prend en paramètre un module de signature :

```
module type HashedType = sig
  type t
  val equal : t -> t -> bool
  val hash : t -> int
end
```

- Le résultat du foncteur est de signature :

```
module type S = sig
  type key
  type 'a t
  val create : int -> 'a t
  val add : 'a t -> key -> 'a -> unit
  [...]
end
```

- Le foncteur est de type :

```
module Make: functor (H : HashedType) -
> S with type key = H.t
```

Foncteurs de la bibliothèque standard

Utilisation du foncteur Hashtbl.Make

```
module NoCaseString = struct
  type t = string
  let equal = fun s1 s2 ->
    String.lowercase s1 = String.lowercase s2
  let hash = fun s -> Hashtbl.hash (String.lowercase s)
end
# module NCSHashtbl = Hashtbl.Make(NoCaseString);;
module NCSHashtbl : sig
  type key = NoCaseString.t
  type 'a t = 'a Hashtbl.Make(NoCaseString).t
  val create : int -> 'a t
  val add : 'a t -> key -> 'a -> unit
  val remove : 'a t -> key -> unit
  val find : 'a t -> key -> 'a
  val find_all : 'a t -> key -> 'a list
  [...]
end
```


Foncteurs de la bibliothèque standard

Utilisation du foncteur Hashtbl.Make

```
# let h = NCSHashtbl.create 17;;
val h : '_a NCSHashtbl.t = <abstr>
# let email = "RmS@gNu.OrG" and name = "Richard M. Stallman";;
# NCSHashtbl.add h email name;;
# h;;
- : string NCSHashtbl.t = <abstr>
# NCSHashtbl.find h "rms@GNU.org";;
- : string = "Richard M. Stallman"
# NCSHashtbl.add h "POTUS@WhiteHouse.gov" "Francis J. Underwood";;
# NCSHashtbl.fold (fun email name r -> (email, name) :: r) h [];;
[("RmS@gNu.OrG", "Richard M. Stallman");
 ("POTUS@WhiteHouse.gov", "Francis J. Underwood")]
# Hashtbl.mem h "RMS@gnu.org";;
Error: This expression has type
         string NCSHashtbl.t = string Hashtbl.Make(NoCaseString).t
       but an expression was expected of type ('a, 'b) Hashtbl.t
```

Plan du cours

Style impératif

Modèle de calcul de la machine de Turing

- **Référence** : cases mémoire
- **Déréférencement** : lecture d'une case mémoire
- **Affectation** : écriture dans une case mémoire
- **Séquence** d'instructions : l'**ordre** d'exécution est primordial, contrairement à l'évaluation des expressions

Effets de bord (*side effect*)

Toute modification d'état non renvoyé par une fonction :

- modification de références globales : **à proscrire**
- impressions (écran, imprimante...)
- communication réseau

Références

Fonction de création : `val ref : 'a -> 'a ref`

Initialisation obligatoire !

```
let a = ref 123
let b = ref 3.14
```

```
let c = ref 'x'
let f = ref (fun x -> 3 * x)
```

Opérateur de déréférencement : `val (!): 'a ref -> 'a`

```
Printf.printf "a = %d\n" !a;;
a = 123
- : unit = ()
```

```
!f 12;;
- : int = 36
```

Opérateur d'affectation : `val (:=) : 'a ref -> 'a -> unit`

```
a := 124
b := !b *. 2.0
```

```
c := 'y'
f := abs
```

Incrémentation, décrémentation `incr, decr : int ref -> unit`

```
incr i; Printf.printf "%d" !i;;
1- : unit = ()
```

```
decr i; Printf.printf "%d" !i;;
0- : unit = ()
```


Portée des liaisons (*scope*)

Une liaison a une **portée** limitée. Elle doit être **la plus locale possible** afin de ne pas pouvoir être utilisée de manière erronée en dehors du contexte.

Liaison globale `let ident = expr;;`

- Définie **en dehors d'une fonction** (ou d'une donnée)
- **Visible** (utilisable) dans toute **la suite** du fichier (éventuellement dans d'autres fichiers)
- **Sauf** si elle est **masquée** par une liaison/paramètre de même nom

À éviter au maximum pour les données **modifiables**

- À réserver pour les **constantes** et les **fonctions**
- Rend le code incompréhensible, peu structuré
- Erreurs très difficiles à corriger
- Code non *réentrant* (une seule instance utilisable à la fois)
- Occupation mémoire permanente

Portée des liaisons

Liaison locale — `let ident = expr1 in expr2`

Visible jusqu'à la fin de la structure syntaxique englobante (liaison, fonction, boucle, conditionnelle...)

```
let x = ref 1;; (* INTERDIT: liaison globale mutable ! *)
let f = fun () ->
  let x = ref 2 in (* masquage *)
  x := 3;;
let g = fun () ->
  x := 4; (* INTERDIT: liaison globale modifiée *)
  let x = ref 5 in
  ();;
let () =
  f (); g (); Printf.printf "%d\n" !x;;
```


Portée des liaisons

Portée statique et limitée à un bloc ($\neq$ Python)

```
def f(x):
    if x % 2 == 0:
        a = 1
    else:
        b = 2
    return x + a
>>> f(2)
3
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in f
UnboundLocalError: local variable 'a' referenced before assignment
```

```
let f = fun x ->
  if x mod 2 = 0 then
    let a = 1 in
    a + x
  else
    let b = 2 in
    b + x
```

```
let f = fun x ->
  let a =
    if x mod 2 = 0 then 1
    else 2 in
  a + x
```

- Ne dépend pas de l'ordre d'exécution
- Tout est vérifié à la compilation
- Beaucoup moins d'opportunités d'écrire n'importe quoi

Structures de contrôle

Instructions

- Séquence : `expr1; expr2; ...; exprn`
- Boucle (répétition)
  - Bornée : `for ident = lb to ub do body done`
  - Non bornée : `while cond do body done`
    À n'utiliser que si **nécessaire** : risque de boucle infinie
- Conditionnelle : `if cond then expr1 else expr2`
- Appel fonctionnel : `f expr1 expr2 ...`
- Exception :
  - Levée : `raise exc`
  - Récupération : `try expr1 with exc -> expr2`

Séquence

Opérateur infixe ';'

`expr1; expr2; ...; exprn`

- Pas besoin de ';' final :

```
let x = ref 2 in
x := !x + 2;
y := !x + 1;
2 * !y
```

- La **valeur renvoyée** par la séquence `expr1; ...; exprn` est celle de la **dernière** expression `exprn`
- Toutes les expressions **sauf la dernière** ne doivent qu'effectuer des **effets de bord** et renvoyer '**()**'. Sinon, un *warning* est émis à la compilation :

```
# 3. ** (1. /. 13.); 12;;
Warning 10: this expression should have type unit.
- : int = 12
```

- Rq. : la liaison locale permet aussi de réaliser une séquence :
  `let _ = expr1 in expr2` ≡ `expr1; expr2`

Boucle bornée

`for ident = int_expr1 to int_expr2 do expr3 done`

- Répétition n **fois** de `expr3`, $n = \texttt{int\_expr2} - \texttt{int\_expr1} + 1$ **fixé** avant d'évaluer `expr3`
- Une boucle bornée **termine** (presque) **toujours**
- **On ne peut pas modifier** `ident` dans `expr3` ($\neq$ C, $\approx$ Python)
- Deux formes **ascendante** et **descendante** :

```
for i = 0 to n - 1 do Printf.printf "%d\n" i done;
for i = n downto 1 do Printf.printf "%d\n" i done;
```

- Le **pas** se gère « manuellement » :

```
for i = 0 to (n-1)/2 do
  t.(2*i) <- 0
  t.(2*i+1) <- 1
done
```

- La **portée** de `ident` est **limitée** à `expr3`

Boucle non bornée

`while bool_expr do expr done`

- Répétition de `expr` **tant que** la condition `bool_expr` est vérifiée :

```
let i = ref 10 in
while !i >= 0 do
  decr i
done
```

- **Une boucle non bornée ne termine pas si elle ne contient pas d'effet de bord** ou ne lève pas d'exception :

```
while true do
  let line = input_line file in (* raise End_of_file *)
  ...
done
```

- **Exclusivement** lorsqu'on ne peut pas utiliser de boucle `for` (bornée)

Conditionnelle sans « else »

if bool_expr then unit_expr

- Si expr1 est de type unit, la branche else est **optionnelle** :

```
if x <> 0 then Printf.printf "%d" x;;
```

- ≡ `if cond then expr else ()`

Appel fonctionnel

f expr1 expr2 ... exprn

- L'application est **prioritaire** sur tous les opérateurs sauf ! :

```
x := f !x (y + 1) + g 2 (h y)
```

`f !x (y + 1)` ≠ `f !x y + 1` ≡ `(f !x y) + 1`

- **Ne jamais prendre de référence en paramètre ou comme valeur de retour**

Plan du cours

Tableaux

Séquence d'éléments

- De taille **quelconque** n **constante** : on ne peut pas modifier la taille d'un tableau après sa création
- **Homogène** : tous les éléments sont de **même type**
- **Indexée** par des entiers de 0 à $n-1$
- Chaque élément est **modifiable**
- Accès en **temps constant**
- Module `Array` : création, itérateur, transformation en liste, tri...

Type

- `int array` : pour un tableau d'entiers
- `'a array` : (prononcer « α array ») pour un tableau d'éléments du même type quelconque `'a` (paramètres de type)

Création

Création avec initialisation obligatoire

- En **extension** :

```
let a = [| 1.5; 2.5; 3.5 |];;
```

- `make: int -> 'a -> 'a array`
  un **élément identique** dans chaque case :

```
let t = Array.make 10 0;;
```

- `init: int -> (int -> 'a) -> 'a array`
  initialisation grâce à une **fonction** de l'index :

```
let carres = Array.init 10 (fun i -> i * i);;
```

Accès

- Accès en **lecture** : `t.(i)`
- **Modification** : `t.(i) <- expr`
- Vérification des bornes

Parcours

Taille — `length: 'a array -> int`

```
# Array.length [|3;2;1|];;
- : int = 3
```

Boucle bornée

```
for i = 0 to Array.length t - 1 do
  s := !s + t.(i)
done
```

Itérateurs

- `iter: ('a -> unit) -> 'a array -> unit`
  application d'une fonction à tous les éléments :

```
Array.iter (fun ti -> s := !s + ti) t
```

- `iteri: (int -> 'a -> unit) -> 'a array -> unit`
  prend également l'index de l'élément en paramètre

Matrices

Tableau de tableaux `'a array array`

- Les valeurs non scalaires ne sont pas dupliquées :

```
let wrong_matrix = fun n m z ->
  Array.make n (Array.make m z);;
# let wm = wrong_matrix 3 2 0;;
val wm : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
# wm.(1).(1) <- 1;;
- : unit = ()
# wm;;
- : int array array = [|[|0; 1|]; [|0; 1|]; [|0; 1|]|]
```

- Il faut utiliser `init` :

```
let make_matrix = fun n m z ->
  Array.init n (fun _ -> Array.make m z);;
let init_matrix = fun n m f ->
  Array.init n (fun i -> Array.init m (fun j -> f i j));;
```


Manipulation de matrices

Trace

```
let trace = fun mat ->
  let t = ref 0 in
  for i = 0 to Array.length mat - 1 do
    t := !t + mat.(i).(i) done;
  !t
# trace [|[|1;2;3|];[|4;5;6|];[|7;8;9|]|];;
- : int = 15
```

Triangle

```
let triangle = fun n f ->
  Array.init n (fun i -> Array.init (i + 1) (fun j -> f i j))
# triangle 5 (fun i j -> i+j);;
- : int array array =
[|[|0|];[|1;2|];[|2;3;4|];[|3;4;5;6|];[|4;5;6;7;8|]|]
```

Manipulation de matrices

Produit

```
let prod = fun a b ->
  let na = Array.length a and ma = Array.length a.(0)
  and nb = Array.length b and mb = Array.length b.(0) in
  assert (ma = nb);
  init_matrix na mb
    (fun i j ->
      let c = ref 0 in
      for k = 0 to ma - 1 do
        c := !c + a.(i).(k) * b.(k).(j)
      done;
      !c)
# prod [|[|1;2;3|];[|4;5;6|]|] [|[|1;2|];[|3;4|];[|5;6|]|];;
- : int array array = [|[|22; 28|];[|49; 64|]|]
```


Autres itérateurs

Un itérateur est souvent préférable à une boucle `for`

Aucun risque de se tromper sur le nombre d'éléments :

- `Array.map f [|x0; ...; xn_1|]` renvoit `[|f x0; ...; f xn_1|]`
- `Array.mapi f [|x0; ...; xn_1|]` renvoit `[|f 0 x0; ...; f (n-1) xn_1|]`
- `Array.fold_right f [|x0; ...; xn_1|] z` renvoit `f x0 (f x1 (f ... (f xn_1 z)...)`
- `Array.fold_left f z [|x0; ...; xn_1|]` renvoit `f (... (f (f z x0) x1) ... xn_1`

Exemples

```
let carres = fun a -> Array.map (fun x -> x*x) a
let produit = fun a -> Array.fold_right (fun x r -> x*r) a 1
let somme = fun a -> Array.fold_left (+) 0 a
let to_list = fun a -> Array.fold_right (fun x r -> x::r) a []
```

Plan du cours

Parce que les programmes manipulent des données

Entrées-sorties

- Lecture et écriture dans des **canaux** (*channel*, parfois appelé *stream*).
- Séquentialité : position implicite (*index*) mise à jour automatiquement
  - Ce qui a été lu ne peut pas être relu
  - Ce qui a été écrit ne peut pas être effacé
- Opérations :
  - ouverture de fichier
  - lecture, écriture
  - déplacement
  - fermeture de fichier

Canaux

Canal

Les lectures et les écritures sont faites par l'intermédiaire d'un *canal*, valeur abstraite correspondant à un fichier ou à un périphérique (écran, clavier...).

- Canal d'entrée : type `in_channel`, par exemple `stdin`
- Canal de sortie : type `out_channel`, par exemple `stdout` et `stderr`

Ouverture et fermeture

- Un fichier doit être **ouvert** pour en obtenir un *canal*.
- La séquence typique d'utilisation sera :
  1. Ouverture : obtention du canal (ou pas : existence, permissions)
  2. Lectures/écritures sur le canal
  3. Fermeture du canal

Ouverture

Core library (`Pervasives`)

```
#open_in;;
- : string -> in_channel = <fun>

#open_out;;
- : string -> out_channel = <fun>
```

Une exception est levée en cas d'erreur :

```
#let ouvre_en_lecture = fun file ->
#  try
#    open_in file
#  with exc ->
#    Printf.printf "%s ne s'ouvre pas en lecture\n" file;
#    raise exc;;
val ouvre_en_lecture : string -> in_channel = <fun>

#ouvre_en_lecture "suchfile";;
suchfile ne s'ouvre pas en lecture
Exception: Sys_error "suchfile: No such file or directory".
```

Écriture formatée dans un canal

Module Printf

- La fonction `fprintf` permet d'écrire dans n'importe quel canal :

```
#Printf.fprintf;;
- : out_channel -> ('a, out_channel, unit) format -> 'a = <fun>
```

- La chaîne de caractères `format` permet d'imprimer une donnée d'un type de base sous différents formats
- La fonction `printf` est un cas particulier de `fprintf` paramétré pour la sortie standard :

```
let printf = fun format -> fprintf stdout format
```


Lecture non formatée

Core library

```
#input_char;;
- : in_channel -> char = <fun>

#input_line;;
- : in_channel -> string = <fun>
```

- Lèvent l'exception `End_of_file` en fin de fichier
- `input_line` alloue la mémoire nécessaire

Copie l'entrée standard jusqu'à la première ligne vide

```
#let copie_jusqu_a_vide = fun () ->
#  let rec encore = fun () ->
#    let l = input_line stdin in
#    if l <> "" then
#      begin Printf.printf "%s\n" l; encore () end in
#  try encore () with End_of_file -> ();;
```

Découpage en mots

Bibliothèque Str (non standard) d'expressions régulières

```
Str.regexp : string -> Str.regexp
Str.split : Str.regexp -> string -> string list
```

Lecture de la date :

```
#let sep = Str.regexp "[ \t]+";;
val sep : Str.regexp = <abstr>

#let read_date = fun line ->
#  match Str.split sep line with
#    day :: month :: year :: _ ->
#       let day = int_of_string day in
#       let month = int_of_string month in
#       let year = int_of_string year in
#       (day, month, year)
#  | _ -> failwith "read_date: unexpected format";;
val read_date : string -> int * int * int = <fun>

#read_date "11\t09  1973";;
- : int * int * int = (11, 9, 1973)
```


Lecture formatée

Module Scanf

Équivalent au `fscanf` en C, mais **les données lues sont passées en paramètre à une fonction** `f`. Le résultat est l'évaluation de `f` :

```
#Scanf.fscanf;;
- : in_channel -> ('a, 'b, 'c, 'd) Scanf.scanner = <fun>

#Scanf.scanf;;
- : ('a, 'b, 'c, 'd) Scanf.scanner = <fun>

#type date = {date: int*string*int; time: int*int};;
type date = { date : int * string * int; time : int * int; }

#let read_date = fun line ->
#  Scanf.sscanf line "%d %s %d %dh%d"
#    (fun j m a h min -> {date = (j, m, a); time = (h, min)});;
val read_date : string -> date = <fun>

#read_date "30 octobre  1961   11h32";;
- : date = {date = (30, "octobre", 1961); time = (11, 32)}
```

Positionnement

Équivalent à `fgetpos` et `fsetpos` en C

Il est possible de connaître la position de l'index de lecture ou d'écriture et de la déplacer n'importe où dans le fichier avec les fonctions :
`pos_in, pos_out, seek_in, seek_out`

```
#pos_in;;
- : in_channel -> int = <fun>

#pos_out;;
- : out_channel -> int = <fun>

#seek_in;;
- : in_channel -> int -> unit = <fun>

#seek_out;;
- : out_channel -> int -> unit = <fun>
```


Fermeture et vidage de mémoire tampon (*buffer flush*)

Fermeture

- Il faut **fermer** un canal après son utilisation :

```
#close_in;;
- : in_channel -> unit = <fun>
#close_out;;
- : out_channel -> unit = <fun>
```

- Tous les fichiers sont fermés automatiquement à la sortie du programme (et les *buffers* vidés)

Buffer

- Toutes les écritures sont *bufferisées* pour ne pas ralentir le programme ni accéder trop souvent au disque
- Il est nécessaire de *flusher* le *buffer* pour synchroniser l'écriture (debuggage, retrait de périphérique...)

```
#flush;;
- : out_channel -> unit = <fun>
```

Plan du cours

Traitement exceptionnel

Rupture du contrôle

- Division par zéro
- Échec du pattern-matching
- Situation inattendue détectée par le programme
- Fin d'un traitement uniforme (cf. I/O)

Exceptions

Une exception est une valeur de type `exn`, prédéfinie ou déclarée :

```
#Failure "Ca c'est mal passe";;
- : exn = Failure "Ca c'est mal passe"

#exception Erreur of int;;
exception Erreur of int

#Erreur 1729;;
- : exn = Erreur 1729
```

Lever une exception

Fonction raise

La levée d'une exception provoque l'abandon de l'évaluation courante :

- pour traiter une erreur
- pour sortir d'une évaluation « profonde », d'une boucle infinie...

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

#raise (Erreur 7);;
Exception: Erreur 7.
```

Traitement d'erreur

```
#let rec dernier = fun l ->
#  match l with
#    [] -> raise (Failure "liste vide")
#  | [x] -> x
#  | x::xs -> dernier xs;;
```


Exceptions et fonctions prédéfinies

Exceptions

- `Match_failure (file, line_number, column_number)`
- `Assert_failure (file, line_number, column_number)`
- `Failure string` : erreur « générale »
- `Invalid_argument string` : par exemple pour l'indexation
- `Not_found` pour les fonctions de recherche
- `End_of_file` pour les fonctions de lectures
- `Division_by_zero` pour l'arithmétique
- `Stack_overflow` pour les dépassements de pile
- `Exit`

Fonctions prédéfinies

```
let invalid_arg =  fun s -> raise (Invalid_argument s)
let failwith = fun s -> raise (Failure s)
```

Exemple : produit optimisé

On lève une exception si un élément est nul

```
#exception Zero;;

#let produit = fun t ->
#  let p = ref 1 in
#  for i = 0 to Array.length t - 1 do
#    if t.(i) = 0 then raise Zero;
#    p := !p * t.(i)
#  done;
#  !p;;


#produit [|1;2;3;0;4;5|];;
Exception: Zero.
```


Récupération d'exception

Poursuite du traitement

- en rattrapant une erreur
- quand un calcul est interrompu par une exception

```
try expression with filtrage de motif
```

Exemple : produit optimisé

```
#let produit_ruse = fun l ->
#  try
#    produit l
#  with
#    Zero -> 0;;
 val produit_ruse : int array -> int = <fun>

#produit_ruse [|1;2;3;0;4;5|];;
- : int = 0
```

Propagation des exceptions

Exception non récupérée

Propagée à l'extérieur du `try with` de fonction en fonction :

```
#exception TropPetit;;

#exception TropGrand;;

#let f = fun x ->
#  if x < 0 then raise TropPetit
#  else if x > 0 then raise TropGrand
#  else failwith "f: nul";;

#let g = fun x ->
#  try (f x) with
#    TropGrand -> Printf.printf "g: %d trop grand\n" x;;

#let h = fun x ->
#  try (g x) with
#    TropPetit -> Printf.printf "f: %d trop petit\n" x;;
```


Propagation des exceptions

Propagation aux fonctions appelantes...

```
#h (-1);;
f: -1 trop petit
- : unit = ()

#h 1;;
g: 1 trop grand
- : unit = ()

#h 0;;
Exception: Failure "f: nul".
```

... jusqu'à sortir du programme

Interception (douteuse) d'exception

Programme qui ne plante jamais

```
let () =
  try
    main ()
  with _ ->
    Printf.printf "Tout va bien..."; main ();;
```

Relais

```
  try
    ...
  with e ->
    Printf.printf "Exception\n"; raise e;;
```


Contrôle et typage

Typage

Attention : les cas « exceptionnels » doivent être de même type que les cas « normaux »

```
exception Resultat of int;;
let f = fun ... ->
  try
    while true do
      ...
      if ... then raise (Resultat n);
      ...
    done;
    0 (* ou failwith "inaccessible" *)
  with Resultat r -> r;;
```

Fonction récursive

Attention à ne pas « empiler » le traitement d'exception à chaque appel récursif