Langage de Programmation 2 (LP2)

RICM3

Cours 1 : Introduction (Base de Caml)

Pascal Lafourcade

Polytech

2009 - 2010

Démo, DOOM


48
AMMO
163%
HEALTH
ARMS
49%
ARMOR
BULL 10 / 200
SHEL 48 / 50
RCKT 20 / 50
CELL 0 / 300
118
AMMO
198%
HEALTH
ARMS
199%
ARMOR
BULL 118 / 200
SHEL 32 / 50
RCKT 5 / 50
CELL 300 / 300

Qui ? Où ? et Quand ?

COURS : Lundi 8h00 - 9h30 Amphi 101

pascal.lafourcade@imag.fr

TD :

- Mardi 16h00 - 18h00, salle 129 : philippe.bidinger@imag.fr
- Mardi 14h00- 16h00, salle 249 : sophie.quinton@imag.fr

TP 3h : Jeudi matin UFR IMA F203 (15 jours)

philippe.bidinger@imag.fr
sophie.quinton@imag.fr

Règles de savoir vivre

Matériel

- Slides online le vendredi avant cours
- Feuille de TD une semaine avant
- Feuille de TP avant
- Page Web LP2

www-verimag.imag.fr/∼plafourc/teaching/LP2_RICM3_2009_2010.php

Note LP2

$$\text{Note finale} = 70\% \text{ Examen} + 10\ \% \text{ CC} + 20\% \text{ TP}$$

1 RV + 1 exo fait.

CC

- Exercices rendus chaque semaine.
- Corrigé par le professeur en début de TD.
- Seulement 5 copies corrigés par semaine choisies au hasard par un programme.
- Correction de 2 exercices par étudiant.
- Note = Max des 2 notes.
- Si absence justifiée pas de correction (prévenir à l'avance).

Possibilité de rendre d'autres travaux pour correction détaillée.

Note LP2

Note finale = 70% Examen + 10 % CC + 20% TP

TP : 4 TPs + 1 TP-Projet

- TP = 60% TP-Projet + 40% TPs
- TPs et TP-Projet par binôme.
- TP-Projet à rendre pour le 22 Avril 2010 minuit par email.
- 4 TPs à rendre une semaine après le TP par email (minuit).
- Note TPs = Max des 2 TPs corrigés
- Si TP pas rendu alors 0.
- Si TOUS les TPs rendus +1 note TP.
- Si absence justifiée pas de correction (prévenir à l'avance).

Possibilité de demander une correction ciblée de parties de TP.

Triche

Ce n'est pas tricher que de

- Regarder le manuel ou le site de Ocaml !
- `http://caml.inria.fr/`
- Demander aux enseignants !
- Envoyer un email aux enseignants !

Tricher c'est :

- Ne pas respecter les deadlines.
- Confondre Travail personnel $\neq$ Travail en groupe.
- Copier, recopier, copier-coller

Politique claire : triche = ZERO pour TOUS.

Votre Travail Personnel

Rappel : Volume de travail personel (6 ETCS = 6*24h)

"J'entends, J'oublie

Je vois, Je me rappelle

Je fais, Je comprends

Confucius

- Amphi sert à apprendre
- TD sert à comprendre
- Projet set à expérimenter

Travail Personnel pour cette UE

- Préparation des TDs suite au Cours (pro-actif)
- Exercices chaque semaine
- TP par groupe
- TP-Projet

Programme du Cours

1. Bases de OCaml (Objective Categorical Abstract Machine Language) ...
2. Récurrence structurelle
3. Evaluation et compléments en Ocaml.
4. Modules, Foncteurs, Input Output
5. Exception, flots
6. Inférence de type (Typage)
7. Polymorphisme, référence, mutable.
8. Lambda Calcul
9. Lambda Calcul
10. Mémoire et point fixe

Programme TD

- listes
- arbres
- tas
- tris
- files
- flots
- foncteurs
- modules
- graphes
- automates
- typage
- ....

Programme TP

1. Introduction à OCAML
2. Code de Huffmann
3. Modules, foncteurs, graphes
4. Analyse lexicale par flots
5. Espace, graphisme : PROJET Doom.

Objectif du cours

Savoir faire

- Changer de paradigme.
- Programmer dans un langage FONCTIONNEL.
- Conception d'algorithmes efficaces et corrects.
- Initiation aux calculs de complexité.

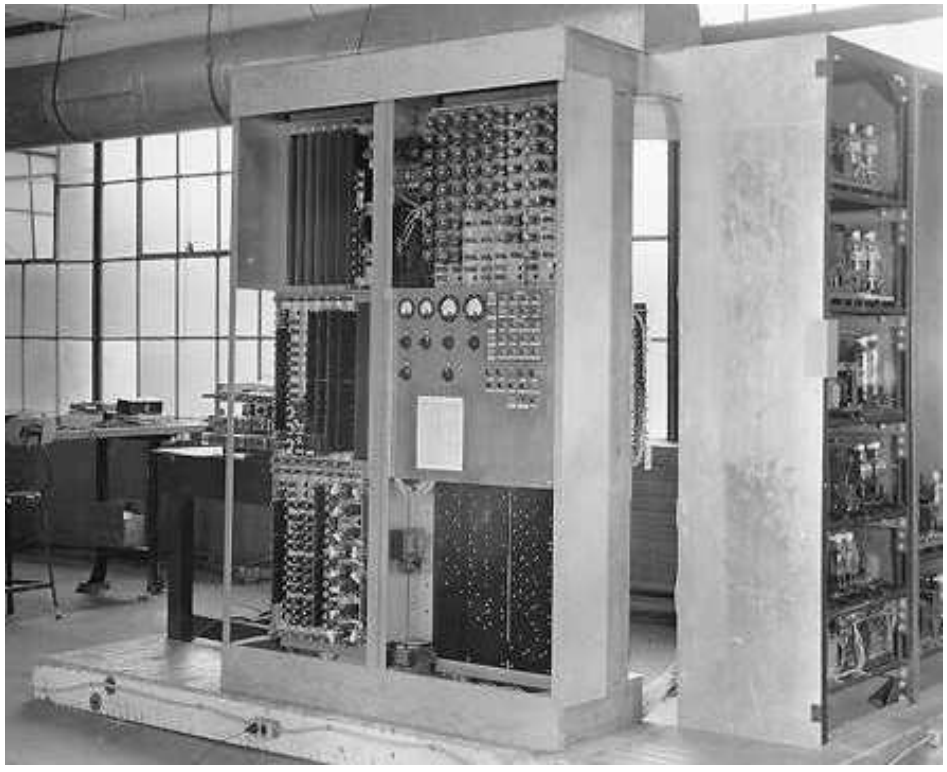
Outils

- Programmation fonctionnelle en Objective Caml.
  (Caml = Categorical Abstract Machine Language)
- Typage
- Raisonnement par récurrence structurelle.
- Preuve de programmes.

Plan

Programmation ?

Ensemble de techniques qui permettent d'écrire un programme informatique pour "discuter" avec la machine.



Ordinateur de Von Neumann 1948.

Historique (I)

- Boulier, arithmétique ...
- 1640 : La Pascaline : calculatrice mécanique de Blaise Pascal
- 1703 : L'arithmétique binaire par Leibniz
- 1801 : Le métier à tisser programmable de J-M Jacquard
- 1820 : La machine à différences de Charles Babbage
- 1843 : Calcul par itération (Ada Lovelace)
- 1847 : L'algèbre de Boole par George Boole

Historique (II)

- 1931 : Incomplétude de Kurt Goedel
- 1936 : Machine de Turing (Alan-Turing)
- 1935 : λ-Calcul (Stephen Kleene, Alonzo Church)
- 1943 : Colossus
- 1948 : Architecture de Von Neumann
- 1948 : Invention du transistor (John Bardeen, William Shockley et Walter Brattain)
- 1956 : Automate de Noam Chomsky
- 1958 : Correspondance de Curry Howard : preuve/programme ou formule/type,
- 1963 : Ordinateurs à circuit intégré
- 1971 : Invention du processeur (IBM 4004 Marcian Hoff).
- 2005 : Ordinateur multi-coeurs.

Différents paradigmes de programmation (styles)

1. Impératif
   Fortran IBM (1954), ALGOL (1958), COBOL (1960), BASIC (1963), PASCAL (1970), C (1970), ADA (1974) ...
2. Fonctionnel
   MIRANDA (1989), HASKELL (1990), MERCURY (1995), OBJECTIVE CAML (1996), Scheme (1970), Lisp (1959), LOGO (1968) ...
3. Objet
   JAVA (1995), C# (2001), VALA (2006), Objective C, EIFFEL (1986), PYTHON (1990), C++ (1985), PHP (1994), SMALLTALK (1972)...
4. Logique Prolog (1972).

ETC : pearl (1987), html (1989), Ruby (1995), csh ...

Programmation impérative

Centrée sur la notion d'assignation

$$x := E$$

- x est un *emplacement mémoire*
- E est une *expression* :
  sa valeur v est temporairement associée à x

x est aussi traditionnellement appelée une *variable*

Modèle de calcul

État

Défini par un n-uplet de données en mémoire
Données décrites *in fine* en termes concrets

Transformations d'état

- État du programme = ⟨mémoire, compteur ordinal⟩
- Mémoire = n-uplet d'associations $\langle x, v \rangle$

Objectif : placer le système dans un état final souhaitable
$\Rightarrow$ connaître tous les chemins d'exécution qui mènent à un état final

Programmation fonctionnelle

Centrée sur la notion d'expression

Toute expression a une *valeur* et un *type*

- Déterminés une fois pour toutes
- La valeur ne dépend pas de l'ordre d'évaluation

Pour le raisonnement :

- on ne se préoccupe que de la *valeur* et du *type*
- raisonnement *par cas* sur les différentes valeurs possibles
  + raisonnement *par récurrence*
- raisonnement *équationnel*
  toute expression peut-être remplacée par une expression égale

Ce que l'on évite

Perturbations

- Dues au contrôle : chemins d'exécution indifférents
- Dues aux données : pas de pointeurs $\Rightarrow$ valeurs mathématiques

Conséquence :

- Meilleure maîtrise d'algorithmes complexes
- Estimation de la complexité (temps de calcul)

Ce que l'on ne perd pas

Efficacité (en Ocaml)

- Interpréteur, compilateur $\longrightarrow$ code-octet
- compilateur $\longrightarrow$ code natif
  efficacité typiquement 1/2 par rapport à C (selon applications)

Possibilité d'utiliser des traits impératifs (à bon escient)

- contrôle séquentiel
- données mutables, pointeurs

Ce que l'on gagne

Efficacité en programmation

- Taille du code typiquement 1/10 du code C équivalent
- Certains problèmes se codent naturellement en fonctionnel
- Gestion mémoire automatique
  - allocation/libération
- Pas de gadget : on se concentre sur l'essentiel

Prix à payer

Effort

- Changement de perspective
- Moins adapté à l'embarqué ou à la programmation système de bas niveau (un peu comme Java)

Support

- Non adossé à une société de grande envergure (ex. Sun)
  (mais communauté très active dans le « libre »)
- Bibliothèques en quantité moyenne
- Trop souvent enseigné à un niveau limité
  $\Rightarrow$ moins de développeurs pointus

Problème *

Écrire une fonction "affine" prenant deux entiers a et b, et renvoyant la fonction $x \longrightarrow ax + b$?

En C : Exercice difficile

Pas de valeur de type : "fonction int vers int"

En Java : Exercice difficile

Idem, mais un peu plus facile car Java est flexible / structuré

En OCaml : Facile

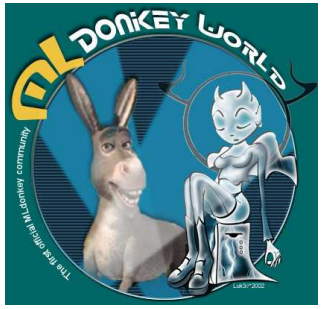
```
let affine a b = (fun x-> a*x+b)
let f = affine 3 2;;
let x = f 2;;(* x = 8 *)
let g = affine 4 5;;
let y = g 1;;(* y =  9*)
```

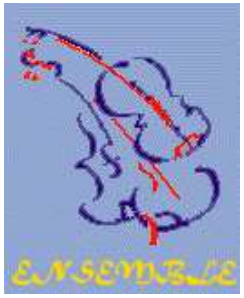
Applications de la programmation fonctionnelle

- Programmation système ou réseau (serveur web)
- Domaine bancaire : analyse financière
- Domaine de prédilection = traitement symbolique :
  - analyse de programmes
  - aide à la démonstration
  - compilation

  $\Rightarrow$ outils de production en milieu industriel
- Calculs numériques avec structures de données complexes
- Milieux scientifiques : mathématiques, physique, biologie ...

Dans la vraie vie





- MLDonkey
- LexiFi
- Ensemble
- Unison Unison
- L'analyseur statique ASTREE 

- SLAM (Microsoft)
- Coq

Caractéristiques de Ocaml

- Typage statique avec inférence de types
- Polymorphisme
- Types sommes et pattern matching (filtrage)
- Gestions des exceptions
- Gestion de la mémoire (Garbage Collector)
- Modules paramétrables (fonctor)
- Compilateur natif et bytecode
- Systéme de classes évolué (objets)
- Préprocesseur expressif et sûr (Camlp4)

Historique

- 1973 ML Milner (tactique de preuves pour le prouveur LCF)
- 1980 Projet Formel à l'INRIA (Gèrard Huet) Catergorical Absrtract Machine (Pierre-Louis Curien)
- 1984-1990 Définition de SML (Milner)
- 1987 Caml (inplémenté en Lisp) Guy =Cousineau, Ascander Suarez (avec Pierre Weis et Michel Mauny)
- 1990-191 Caml Light par Xavier Leroy (et Damien Doligez pour la gestion de la mémoire)
- 1995 Caml Special Light
- 1996 Ocaml (Xavier Leroy, Jêrome Vouillon, Didier Rémy, Michel Mauny)

Types de base

- int, constantes : 42 -12 opérateurs : + - * / mod
- float, constantes : -3.14 0. 1.3e-11 opérateurs : +. -. *. /.
- char, constantes : 'a' 'B' char_of_int, int_of_char
- string, constantes : "abc" opérateur de concaténation : ^
- bool, constantes : true false
  opérateur de concatenation : &, &&, or, ||, not
- α Type polymorphe
- unit Type spécial avec un seul élement ()

Expression conditionnelle

- "if b then e_1 else e_2"
- évaluation en fonction de b

Exemple : (if $5 < 9$ then 21 else 3) *2 ; ;

- (if $5 < 9$ then 21 else 3) *2 ; ;
- (if true then 21 else 3) *2
- 21*2
- 42

Remarque :
e1 && e2 : e2 n'est pas évaluée si e vaut false
e1 || e2 : e2 n'est pas évaluée si e1 vaut vrai

Listes

- Liste vide []
- ajout en tête : :

Exemple

```
# 'O' : :('c' : :('a' : :('m' : :('l' : :[])))) ; ;
- : char list = ['O' ;'c' ;'a' ;'m' ;'l']
```

comparaison : ordre lexicographique

```
# ['a' ;'b'] < ['a' ;'a'] ; ;
- : bool = false
```

Précision : Deux notions de variable

Mathématiques $A \underline{\underline{\text{déf}}} 2\sin x + 3x$ **Informatique**

Variable en mathématiques

La valeur de A *dépend* de la valeur de x :

- la valeur de x est fixée (même si elle peut être inconnue)
- si x varie, en fonction du temps t cela est explicité : $x(t)$
  Ex. : si $x(t) = 5t^2 - 1$ alors $A(t) = 2\sin(5t^2 - 1) + 15t^2 - 3$

Variable en informatique

- le temps t reste implicite (cadencement par l'horloge)
- t est discret
- $x(t)$ dépend des valeurs en mémoire à l'instant précédent : $x(t-1)$, $y(t-1)$, etc.

Variables *

Définition de variable : let

Variable = association nom/valeur

```
# let x = 53 ; ;
val x : int = 53

# let y = x - 11 ; ;
val y : int = ?42
```

Explications

- let y = x - 11 ; ;
- let y = 53 - 11 ; ;
- let y = 42 ; ;
- val y : int = 42

Redéfinition de variables *

Variable = nom donné à une valeur
Variable ≠ nom donné à une zone mémoire (en C et Java)

Exemple

```
# let x = 53 ; ;
val x : int = 53
# let y = x - 11 ; ;
val y : int = 42
# let x = 2 ; ;
val x : int = 2
# y ; ;
- : int = ? -9 ou 4242
```

Important *

Casse

```
# let eta = 0 ; ;
val eta : int = 0
# let eTa =1 ; ;
val eTa : int = 1
# eta = eTa ; ;
- : bool = false
```

Ocaml est case-sensitive

Fonctions *

Notation
mathématique : $\lambda x.f(x)$ Ocaml : fun $x \rightarrow f(x)$

Exemple

```
# fun x -> x <> 0 ; ;
- : int -> bool = <fun>
# (fun x -> x <> 0) 42 ; ;
- : bool = true
```

```
# let estnonnul = (fun x -> x <> 0) ; ;
val estnonnul : int -> bool = <fun>
# estnonnul 0 ; ;
- : bool = false
```

Fonctions à plusieurs arguments

Point de vue au premier ordre (usuel en maths)

Fonction à n arguments = fonction à 1 argument : n-uplet

Exemples :

- ds $= \sqrt{dx^2 + dy^2 + dz^2 - 3.14dt^2}$

  ds : float $\times$ float $\times$ float $\times$ float $\longrightarrow$ float

  ```
  # let ds (dx, dy, dz, dt) =
     sqrt (dx ** 2. +. dy ** 2. +. dz ** 2. -. (3.14 *. dt) **
  2.) ; ;
  ```

  1 argument : quadruplet

Fonctions à plusieurs arguments "Curryfication"

Point de vue à l'ordre supérieur (préféré en prog. fonct.)

- Fonction à 0 argument = constante
- Fonction à $n+1$ arguments =
  fonction à 1 argument qui rend une fonction à n arguments

Exemples

plus 3 2 se lit (plus 3) 2

plus : int → int → int qui se lit int → (int → int)

ds : (float → (float → (float → (float → float))))

```
# let ds dx dy dz dt =
    sqrt (dx ** 2. +. dy ** 2. +. dz ** 2. -. (c *. dt) ** 2.) ; ;
```

Besoins contradictoires

Essence du typage

Les arguments fournis à une fonction doivent avoir un sens : ne pas mélanger entiers, chaînes, booléens, n-uplets, fonctions...

Mais on a souvent besoin de mélanger

- protocoles
- lexèmes, structures grammaticales

Fausse solution : union (casse le typage)
Solution compliquée : types avec discriminants
Bonne solution : union disjointe = somme

Types somme *

Exemple

type ecb =
    Ent of int | Ch of string | Boofl of bool × float
Ent, Ch, Boofl = *constructeurs* du type

Valeurs : Ent (3), Ch ("azerty"), Boofl (true, 3.14)
Étant donnée une valeur, l'examen du constructeur permet de déterminer le type d'origine (ex. int, string, bool × float)

Sommes : exemples dégénérés

Énumération `# type couleur = Rouge | Vert | Bleu | Jaune`
`# type boule = top | bottom ; ;`
Syntax error
`# type boule = Top| Bottom ; ;`
Booléen `type bool = true | false`
Unité `type unit = ()`

Application : Jeu de carte

Définir une type somme afin de manipuler un jeu de 32 cartes pour pouvoir jouer en Ocaml ?

Solution

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

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

let dixtrefle = Petite (10, Trefle) ; ;
let ascoeur = As Coeur ; ;
```

! Impossible de définir un type entier entre [2..10]

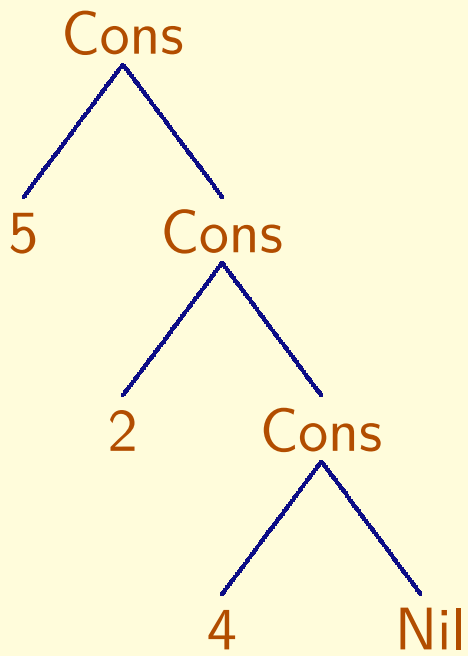
! cela rendrait le typage indécidable.

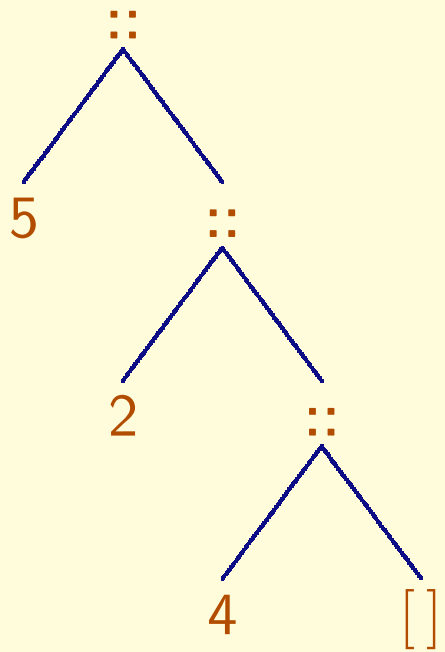
Type somme récursif : Listes *

```
type liste_ent =
  | Nil
  | Cons of int * liste_ent
```

Exemple : [5 ; 2 ; 4]

Cons (5, Cons (2, Cons (4, Nil)))





Analyse par cas *

type ecb = Ent of int | Ch of string | Boofl of bool × float

Filtrage

Soit v une *valeur* de type ecb

match v with
| Ent (n) $\rightarrow e_1$
| Ch (s) $\rightarrow e_2$
| Boofl (b, x) $\rightarrow e_3$

Les expressions e_i à droite de « $\rightarrow$ » doivent avoir le même type

Filtrage = analyse par cas + décomposition

type ecb = Ent of int | Ch of string | Boofl of bool × float

Décomposition

```
match V with
| Ent (n) →... n ...
| Ch (s) →... s ...
| Boofl (b, x) →... x ... b ...
```

Les expressions de même type à droite de « → », sont toujours bien définies :

- la première dans un environnement augmenté de n : int
- la seconde dans un environnement augmenté de s : string
- la troisième dans un environnement augmenté de b : bool et de x : float

Exemple : listes

```
type liste_ent =
  | Nil
  | Cons of int * liste_ent

match l with
  | Nil → true
  | Cons (x, l) → false

match l with
  | [] → true
  | x :: l → false
```

Exemple : booléens

match b with true $\longrightarrow$ e_1 | false $\longrightarrow$ e_2

est juste une autre syntaxe pour

if b then e_1 else e_2

Aujourd'hui

- Programmation fonctionnelle
- Types et opérateurs de base (int, char ...)
- Variables (let)
- Fonction (fun)
- Types somme (type = ... of ...)
- Filtrage (match)

Prochaine fois

- Récurrence
- Filtrage
- Preuves
- Fonctions d'ordre supérieur

Merci de votre attention.

Questions ?

Bibliographie

- Xavier Leroy et al. Objective Caml manual `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`
- The book Developing Applications With Objective Caml (by Emmanuel Chailloux, Pascal Manoury and Bruno Pagano)
- Approche fonctionnelle de la programmation, Michel Mauny , Guy Cousineau
- Pierre Weis and Xavier Leroy. Le langage Caml. Dunod, 1999.
- Apprentissage de la programmation avec OCaml Catherine Dubois & Valérie Ménissier-Morain
- Philippe Nardel. Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml. Vuibert, 2005
- Louis Gacogne. Programmation par l'exemple en Caml. Ellipse, 2004