

Prise en main de Caml

Partie A

Généralités

# CamI

- ML = MetaLanguage (Milner)
- langage fonctionnel (comme Lisp)
- inférence de type
- CamI = CAML = Categorical Abstract Machine Language

# Ressources

- Le langage Caml (Weis & Leroy)
- Approche... (Veigneau)
- Web Caml : `pauillac.inria.fr`
- Web perso : `bruno.maitresdumonde.com/optinfo`

# Démarrage de l'atelier

1. Allumer l'ordinateur
2. Ouvrir une session
3. Double-clic sur l'icône du chameau
4. Choisir un nom de fichier finissant par `.ml`

# L'atelier

1. Emacs (éditeur programmable, conçu par Richard Stallman)
2. Mode Tuareg (Albert Cohen)
3. Interpréteur `camllight` (INRIA)
4. Le tout sous Linux !

# Extinction de l'atelier

1. Quitter Emacs (Ctrl-X Ctrl-C)
2. Fermer la session (Ctrl-Alt-Backspace)
3. Ou : éteindre l'ordinateur (selon convenance)

# Dialoguer avec l'atelier

- Saisie du texte
- Enregistrer (Ctrl-X Ctrl-S)
- Compiler (Ctrl-C Ctrl-B)

Prise en main de Caml

Partie B

Premiers pas

# Calculs avec des entiers (1)

- Opérateurs : +, -, \*, /, mod
- Relations : =, >, <, <=, >=, <>
- min, max
- Parenthèses

## Calculs avec des entiers (2)

- Intervalle représenté
- Calculer  $32768^2$  puis  $32768^2 - 1$
- Explications : écriture en binaire
- Architecture 32 vs. 64 bits

# Noms et liaisons

- Syntaxe des noms
- Syntaxe d'une liaison simple
- Syntaxe d'une liaison multiple; sémantique, *threads* multiples
- Liaison locale: `let .. in ..`

# Définition d'une fonction

- Type des opérations sur entiers
- Un exemple : carré
- Composer deux fonctions
- Syntaxe de l'application de fonction
- Syntaxe de la définition de fonction

# Couples, $n$ -uplets

- Couples : syntaxe, définition
- Fonctions `fst` et `snd`
- Rédigeons ces fonctions !
- $n$ -uplets (*tuples*)

# Le type `bool`

- Les valeurs possibles
- Opérateurs `&`, `or`, `not`
- Règle d'évaluation paresseuse
- Construction `if .. then .. else ..`
- Quelques écritures maladroites !

# Prise en main de Caml

## Partie C

Chaînes de caractères

# Type char

- sert à représenter les caractères
- codage textuel : 'a'
- caractères spéciaux : '\'', '\\', ...
- codage numérique : '\065'
- char\_of\_int (code entre 0 et 255 inclus)
- int\_of\_char

# Type string (1)

- chaînes de caractères
- "abc def"
- caractères spéciaux: "abc\"def\""
- string\_length
- s.[i], l'indexation commence à 0

## Type string (2)

- `"a".[0]` est égal à ... ?
- `s.[i] <- c`
- `make_string`
- concaténation: `s1 ^ s2`

# Miroir d'une chaîne (v1)

Rédigeons une fonction `miroir`.

Son type sera `string -> string`:

```
let miroir s =  
  let n = string_length s in  
  let s' = make_string n s.[0] in  
  for i = 0 to n - 1 do  
    s'[i] <- s.[n-1-i]  
  done;  
  s' ;;
```

# Fonctions d'impression (1)

- `print_char`
- `print_string`
- `print_int`
- `print_newline`

## Fonctions d'impression (2)

Exemple :

```
let a = 3 and b = 4 in
  print_string "la somme des carrés de ";
  print_int a;
  print_string " et ";
  print_int b;
  print_string " est égale à ";
  print_int (a*a+b*b);
  print_newline() ;;
```

## Fonctions d'impression (3)

La fonction `print_xxx` est de type `xxx -> unit`.

Elle rend `()`, constante du type `unit`.

Leur emploi fait sortir de la programmation fonctionnelle pure.

Il n'y a pas de `print_bool`: rédigeons-le!

## Rôle du $;$

$;$  est l'opérateur de séquençement.

$e ; f$  s'interprète comme suit :

- l'expression  $e$  est évaluée
- le résultat  $\hat{e}$  est jeté
- l'expression  $f$  est évaluée
- le résultat  $\hat{f}$  est celui de  $e ; f$

# Notion d'effet

Le résultat  $\hat{e}$  doit être (), sinon Caml râle!

Évaluer  $e$  réalise un «effet» :

- lecture : `read_int`
- impression : `print_string`
- modification : `s.[3] <- 'z'`

# Miroir d'une chaîne (v2)

Jean-Maurice Maladroit calcule le miroir sur place :

```
let miroir_sur_place s =  
  let n = string_length s in  
  for i = 0 to n - 1 do  
    s.[i] <- s.[n-1-i]  
  done;;
```

1. où est l'erreur ?

2. comment la corriger ?

# Prise en main de Caml

## Partie D

Types énumérés, filtrage

# Types énumérés

- Déclaration de type énuméré
- Affectations de valeurs
- Filtrage sur un type énuméré
- Exemple : `signum`
- Exemple : mélange de couleurs

# Déclaration de type énuméré

```
type signe = Positif | Négatif | Zéro ;;
```

```
type couleur = Rouge | Blanc | Rose  
| Bleu | Violet | Jaune | Orange ;;
```

Un type énuméré est un ensemble fini, dont on nomme chaque élément !

# Affectations de valeurs

```
let c1 = Blanc and c2 = Rouge in ...
```

```
let drapeau = (Bleu,Blanc,Rouge) in ...
```

# Filtrage sur un type énuméré

```
let température_couleur = function  
  | Rouge | Jaune | Orange -> "chaud"  
  | Bleu | Vert -> "froid" ;;
```

Remarque: le filtrage n'est pas exhaustif!

## Exemple : `signum`

```
let signum = function
  | n when n > 0 -> Positif
  | n when n < 0 -> Négatif
  | _ -> Zéro ;;
```

Remarque : le filtrage est exhaustif, car le dernier motif «attrape tout».

On aurait pu écrire le dernier cas :

```
| n -> Zéro ;;
```

# Mélange de couleurs (1)

```
let mélange = fonction
  | (Rouge,Blanc) | (Blanc,Rouge) -> Rose
  | (Rouge,Bleu) | (Bleu,Rouge) -> Violet
  | (x,y) when x=y -> x
  | _ -> failwith "mélange inconnu" ;;
```

Remarque: `failwith` lève une exception.

# Mélange de couleurs (2)

Observez :

```
mélange(Bleu,Blanc);;
```

```
2/0;;
```

```
let x = 3 and y = 0 in
```

```
try
```

```
  (x/y,x mod y)
```

```
with _ -> print_string "Vous exagérez" ;;
```

## print\_bool avec un filtre

```
let print_bool = function  
  | true -> print_string "vrai"  
  | _ -> print_string "faux" ;;
```