Introduction à CAML

Les nombres entiers

```
# 1 ;;
- : int = 1

# 1 + 2 ;;
- : int = 3

# 9 / 2 ;;
- : int = 4

# 2 < 5;;
- : bool = true
```

Les opérations sur les entiers

Opérations arithmétiques : $+$, `-`, `*`, $/$, `mod`.

Opérations de comparaison : `=`, `<>`, `<`, `<=`, `>`, `>=`.

Les nombres flottants

```
# 2.0 ;;
- : float = 2

# 1.1 +. 2.2 ;;
- : float = 3.3

# 9.1 /. 2.2 ;;
- : float = 4.13636363636

# 2. /. 0. ;;
- : float = inf

# -. 3;;
- : int = -3
```

```
# 2. ** 2.;;
- : float = 4

# 2 = 2.;;
This expression has type float but is here used with type int
# 2. + 4;;
This expression has type float but is here used with type int

# ceil(3.4);;
- : float = 4

# floor(3.4);;
- : float = 3

# ceil(-. 3.4);;
- : float = -3
```

```
# floor(-. 3.4);;
- : float = -4
```

Les opérations sur les flottants

Opérations de comparaison : `=`, `<>`, `<`, `<=`, `>`, `>=`.

D'autres opérations : `ceil`, `floor`, `sqrt`, `exp`, `log`, `sin`,

...

Les caractères

```
# 'a';;
- : char = 'a'
# '1';;
- : char = '1'
# int_of_char('B');;
- : int = 66
# int_of_char('b');;
- : int = 98
int_of_char('\200');;
- : int = 200
```

Les chaînes

```
# "bonjour";;
- : string = "bonjour"
# "abc"^"def";;
- : string = "abcdef"
# "";;
- : string = ""
# string_of_int(1987);;
- : string = "1987"
# int_of_string "123" + 2;;
- : int = 125
```

Les booléens et ses opérateurs

Les constates

```
# true ;;
- : bool = true
# false ;;
- : bool = false
```

L'opérateur de négation

```
# not true ;;
- : bool = false
# not false ;;
- : bool = true
```

L'opérateur de disjonction

```
# true or true ;;
- : bool = true
# true or false ;;
- : bool = true
# false or true ;;
- : bool = true
# false or false ;;
- : bool = false
```

L'opérateur de conjonction

```
# true & true;;
- : bool = true
# true & false ;;
- : bool = false
# false & true;;
- : bool = false
# false & false ;;
- : bool = false
```

Les paires et les n-uplets

- Ça permet de rassembler plusieurs éléments de type différent.
- Pour les paires, on peut accéder à ses composantes par des fonctions de projection `fst` et `snd`.
- Pour les n-uplets de la forme `(a1,..,an)` on peut construire n fonctions de projection pour accéder à chaque composante `ai`.

```
# (2,'a');;
- : int * char = 2, 'a'
# fst(2,'a');;
- : int = 2
# snd(2,'a');;
- : char = 'a'
# (2, ('a','b'));;
- : int * (char * char) = 2, ('a', 'b')
# fst(snd(2, ('a','b')));;
- : char = 'a'
# (1,2,3,4);;
- : int * int * int * int = 1, 2, 3, 4
```

Structures conditionnelles simples

Syntaxe Si b, e_1 et e_2 sont des expressions alors `if` b `then` e_1 `else` e_2 est une expression.

Condition de bonne formation Si b est une expression dans le domaine des booléens et e_1 et e_2 sont des expressions dans le domaine t alors `if` b `then` e_1 `else` e_2 est une expression bien formée dans le domaine t.

Exemples

```
# if 3=4 then 0 else 5;;
- : int = 5
# if 3=4 then "aa" else "bb";;
- : string = "bb"
# (if 3=4 then 0 else 5) + 8;;
- : int = 13
# if (3=4) or (5<6) then 0 else 5;;
- : int = 0
# if 3=4 then 0 else "aa";;
This expression has type string but is here used with type int
```

Structures conditionnelles multiples

- C'est un outil de discrimination selon la forme d'un ou plusieurs éléments.
- Très confortable pour la programmation récursive.
- Permet l'écriture de programmes lisibles.

Syntaxe

$$
\begin{array}{ll}
\texttt{match}\ (e_1, \ldots, e_k)\ \texttt{with} & \\
\quad (m_1^1, \ldots, m_1^k) & -> \mathit{expr}_1 \\
\quad \mid (m_2^1, \ldots, m_2^k) & -> \mathit{expr}_2 \\
\quad \mid \vdots & -> \\
\quad \mid (m_n^1, \ldots, m_n^k) & -> \mathit{expr}_n \\
\quad \mid \_ & -> expr
\end{array}
$$

est une expression équivalente au conditionnelle simple :

$$
\begin{array}{l}
\texttt{if}\ \ (e_1, \ldots, e_k) = (m_1^1, \ldots, m_1^k) \\
\quad \texttt{then}\ expr_1 \\
\quad \texttt{else if}\ (e_1, \ldots, e_k) = (m_2^1, \ldots, m_2^k) \\
\qquad\quad \texttt{then}\ expr_2 \\
\quad \vdots \\
\qquad\quad \texttt{if}\ (e_1, \ldots, e_k) = (m_n^1, \ldots, m_n^k)\ \texttt{then}\ expr_n\ \texttt{else}\ expr
\end{array}
$$

Exemple

```
# match (4+3+20) with
    0 -> "a"
  | 1 -> "b"
  | _ -> "d";;
- : string = "d"
# match (4+3+20) with
    0   -> "a"
  | 'a' -> "c"
  | _   -> "d";;
This pattern matches values of type char
but is here used to match values of type int
```

Définition de valeurs

- Déclarations globales simples
- Déclarations globales simultanées
- Déclarations locales simples
- Déclarations locales simultanées

Déclarations globales simples

Déclaration d'une variable dans le programme jusqu'à sa prochaine déclaration globale.

Syntaxe `let` $var = expr_1$

```
# let x = 3;;
val x : int = 3
# let y = x +2 ;;
val y : int = 5
# let x = 5;;
val x : int = 5
# let y = x +2 ;;
val y : int = 7
```

Déclarations globales simultanées

Syntaxe `let` $var_1 = expr_1 \ldots$ `and` $\ldots\ var_n = expr_n$

```
# let x =3 and y = 5;;
val x : int = 3
val y : int = 5
# x+y;;
- : int = 8
# let x =3 and x = 5 ;;
This variable is bound several times in this matching
# let x=3 and y = x;;
Unbound value x
```

Déclarations locales simples

Une variable est visible uniquement dans l'expression qui l'utilise.

Syntaxe `let` $var = expr_1$ `in` $expr_2$

```
# let x =3 in x*x;;
- : int = 9
# (let x =3 in x*x) + 20 ;;
- : int = 29
# let x = 3 in let y = 5 in x+y;;
- : int = 8
# let x = 3 in let y = x*y in x+y;;
- : int = 18
```

Déclarations locales simultanées

Syntaxe `let` $var_1 = expr_1 \ldots$ `and` $\ldots\ var_n = expr_n$ `in` $expr$

```
# let a = 4. and b = sqrt(a) in a *. b ;;
Unbound value a
# let a = 4. and b = sqrt(9.) in a *. b ;;
- : float = 12
# let a = 4. and b = sqrt(a) in a *. b ;;
Unbound value a
```

Les fonctions avec nom

Syntaxe `let` $nom(par_1, \ldots, par_n) \; = \; expr$

- C'est une déclaration *globale* de la fonction *nom* ayant comme paramètres $par_1, \ldots, par_n$ et définie par l'expression *expr*.
- Il faut respecter son *arité* pour son utilisation.
- Il faut respecter son *type* pour son utilisation.

```
# let min(x,y) = (x+y-abs(x-y))/2;;
val min : int * int -> int = <fun>
# min(2,5);;
- : int = 2
# min(10,-3);;
- : int = -3
# min(4);;
This expression has type int but is here used with type int * int
# min(3.,5.);;
This expression has type float * float but is here used with type int* int
```

Différentes solutions d'un problème

```
# let min4_v1(x,y,z,w) =
       let m=min(x,y) and n = min(z,w) in min(m,n);;
val min4_v1 : int * int * int * int -> int = <fun>
# let min4_v2(x,y,z,w) =
       let m=min(x,y) in let n=min(m,z) in min(n,w);;
val min4_v2 : int * int * int * int -> int = <fun>
# let min_v3(x,y,z,w) = min(min(x,y),min(z,w));;
val min_v3 : int * int * int * int -> int = <fun>
```

Les fonctions sans nom

Syntaxe `function` $(par_1, \ldots, par_n) -> \ expr$

```
# function(x,y) -> x*y /2;;
- : int * int -> int = <fun>
# (function(x,y) -> x*y /2) (3,4);;
- : int = 6
# function(x) -> (function(y) -> x*y/2);;
- : int -> int -> int = <fun>
# (function(x) -> (function(y) -> x*y/2))(5);;
- : int -> int = <fun>
# (function(x) -> (function(y) -> x*y/2))(5)(7);;
- : int = 17
```

Utilisation de fonctions sans nom

```
# function(x,y) -> if x< y then x+1 else y;;
- : int * int -> int = <fun>
# let f = function(x,y) -> if x< y then x+1 else y;;
val f : int * int -> int = <fun>
# f(3,8);;
- : int = 4
# f(8,2);;
- : int = 2
```

Syntaxe alternative

Syntaxe `function` $par_1 \ldots par_n \; -> \;\; expr$

```
# fun x y -> x*y /2;;
- : int -> int -> int = <fun>
# (fun x y -> x*y/2) 3 4 ;;
- : int = 6
# fun x -> fun y -> x*y/2;;
- : int -> int -> int = <fun>
# (fun x -> fun y -> x*y/2) 5 ;;
- : int -> int = <fun>
# (fun x -> fun y -> x*y/2) 5 7 ;;
- : int = 17
```

Polymorphisme

Motivations :

- Décrire le comportement d'une opération de façon générale, i.e. de façon indépendante de la nature (ou type) de ses paramètres.
- Employer une même opération avec plusieurs types différents. Ceci rend le programme plus claire et plus succint.
- Faciliter la réutilisation.
- Faire porter le même nom à des opérations qui se "ressemblent" d'un point de vue sémantique.

Exemple :

```
# milieu_trois(1,2,3);;
- : int = 2
# milieu_trois('a','b','c');;
- : char = 'b'
# milieu_trois("Dupont","Leblanc","Legrand");;
- : string = "Leblanc"
```

Solution en CAML

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

`'a, 'b, 'c` représentent trois types `quelconques`.

Un deuxième exemple

```
# dup(3);;
- : int * int = 3, 3
# dup('a');;
- : char * char = 'a', 'a'
# dup(2,2);;
- : (int * int) * (int * int) = (2, 2), (2, 2)
dup("le","li","la");;
- : (string * string * string) * (string * string * string) =
("le", "li", "la"), ("le", "li", "la")

let dup(x) = (x,x);;
val dup : 'a -> 'a * 'a = <fun>
```

Un troisième exemple

```
# let composition(f,g)(x) = f(g(x));;
val composition : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>

# composition((fun x -> x+4),(fun y -> y*5))(8);;
- : int = 44

# composition((fun y -> y*5),(fun x -> x+4))(8);;
- : int = 60

# composition(dup,dup)(3);;
- : (int * int) * (int * int) = (3, 3), (3, 3)
```

Définition de Types

L’ordre supérieur