
Introduction à OCAML

Plan

L'interpréteur interactif Ocaml

Les types de base, et leurs opérations :

int, float, char, string, bool

Les n -uplets

Conditionnelles :

if ... then ... else et match ... with ...

Déclarations :

let ... in ...

Fonctions : avec et sans nom, différence entre fun et fonction
fonctions d'ordre supérieure

Polymorphisme : réutiliser le code

Types utilisateur :

types énumérés, types avec étiquettes, types paramétriques

Types de Données Abstraites : notions et exemples

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. ;;
- : float = -3

# 2. ** 3. ;;
- : float = 8

# 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

# (* mais on peut convertir en utilisant les fonctions ...*)

#truncate 3.4;;
- : int 3

#float 3;;
- : float 3.0

# 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 de caractères

```
# "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 constantes

```
# 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
```

Notation équivalente :

$$e_1 \text{ or } e_2 \equiv e_1 \mid e_2$$

L'opérateur de conjonction

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

# true & false ;;
- : bool = false

# false & true;;
- : bool = false

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

Notation équivalente :

$$e_1 \text{ \& } e_2 \equiv e_1 \ \&\& \ e_2$$

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 même 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

FILTRAGE/PATTERN-MATCHING

- Outil de discrimination selon la **forme** d'un ou plusieurs éléments (valeurs).
- Très **confortable** pour la programmation réursive.
- Permet l'écriture de programmes **lisibles**.

Syntaxe

```

match e with
  m1 -> expr1
| m2 -> expr2
| : ->
| mn -> exprn
| _ -> expr

```

avec m_1, m_2, \dots, m_n du même type et
 $expr_1, expr_2, \dots, expr_n, expr$ du même type.
est une expression équivalente au conditionnelle simple :

```

if e = m1
  then expr1
  else if e = m2
    then expr2
    :
    if e = mn then exprn else expr

```

Exemples

```

# 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

```
# match ( (true & false) || true, not false || true) with
  (true, true) -> true
| (true, false) -> false
| (false, true) -> false
| (false, false) -> false;;
- : bool = true
```

```
# match ( (true & false) || true, not false || true) with
  (true, true) -> true
| (_,_) -> false;;
- : bool = true
```

```
# match ( (true & false) || true, not false || true) with
  (true, _) -> true
| (_,false) -> false
| _ -> false;;
- : bool = true
```

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 = expr1`

```
# 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 var1 = expr1 ... and ... varn = exprn`

```
# 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 = expr1 in expr2`

```
# 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 and y= x*y in x+y;;
Unbound value x
```

```
# let x = 3 in let y = 3*y in x+y;;
Unbound value y
# let x = 3 in let y = 3*5 in x+y;;
- : int = 18
```

Déclarations locales simultanées

Syntaxe `let var1 = expr1 ... and ... varn = exprn 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
```

Les fonctions avec nom

Syntaxe `let nom par1 ... parn = expr`

- C'est une déclaration *globale* de la fonction *nom* ayant comme paramètres la liste *par₁, ..., par_n* et définie par l'expression *expr*.
- Il faut respecter son *type* pour son utilisation.

```
# let a_au_milieu x y = x^"a"^y;;
val a_au_milieu : string -> string -> string = <fun>
# a_au_milieu "bbbb" "cccc";;
- : string = "bbbbaaaaa"
# a_au_milieu "bbbb" 4;;
This expression has type int but is here used with type string
```

Les fonctions avec nom : cas particulier

Syntaxe `let nom par1 = expr`

La fonction *nom* possède **un seul** paramètre *par₁*

```
# let deux_fois x = x^x;;
val deux_fois : string -> string = <fun>
# deux_fois "aaa";;
- : string = "aaaaaaa"
# deux_fois 3;;
This expression has type int but is here used with type string
```

Les fonctions avec nom : cas particulier

Syntaxe `let nom (par11, ..., par1n) = expr`

- La fonction *nom* possède **un seul** paramètre qui est un *n*-uplet de la forme $(par_1^1, \dots, par_1^n)$.
- Lors de l'utilisation de la fonction *nom* il faut respecter son *type*, c'est à dire, il faut respecter l'*arité* *n*.

```
# 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 `fun par1 ... parn -> expr`

C'est une fonction qui s'applique à **une suite** de *n* arguments.

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

Les fonctions sans nom : cas particulier d'un seul argument

Syntaxe : `fun par -> expr`

Fonction sans nom qui s'applique à **une suite d'un seul** argument.

```
# fun x -> x *2;;
- : int -> int = <fun>
# (fun x -> x *2) 4 ;;
- : int = 8
# fun (x,y) -> x*2*y;;
- : int * int -> int = <fun>
# (fun (x,y) -> x*2*y) 2 ;;
This expression has type int but is here used with type int * int
# (fun (x,y) -> x*2*y) (2,3);;
- : int = 12
```

Définition par cas sur un seul argument

Syntaxe `function par -> expr`

```
# function x -> x/2;;
- : int -> int = <fun>
# (function x -> x/2) 3;;
- : int = 1
```

Mais aussi :

```
# function 0 -> true | n -> false;;
- : int -> bool = <fun>
# (function 0 -> true | n -> false) 3;;
- : bool = false
```

Cas particulier

Syntaxe `function (par1,...,parn) -> expr`

L'argument **unique** est un **n-uplet**.

```
# function (x,y) -> x*y /2;;
- : int * int -> int = <fun>
# (function (x,y) -> x*y /2) (3,4);;
- : int = 6
# (function (x,y) -> x*y /2) 3 4 ;;
This function is applied to too many arguments
```

Utilisation de fonctions sans nom

```
# let f1 = fun x y -> x * y/2;;
val f1 : int -> int -> int = <fun>
# f1 3 ;;
- : int -> int = <fun>
# f1 3 4 ;;
- : int = 6
```

Ou alternativement...

```
# 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
```

L'ordre supérieur

Motivations :

- Manipuler des fonctions comme des données : des opérations qui reçoivent des fonctions comme argument et qui retournent des fonctions comme résultat.
- Réutilisation du code : certaines fonctions peuvent être réutilisées dans de nombreuses situations.

Fonctions en tant qu'argument d'un paramètre

```
# let app(f,y) = (f y) + y;;
val app : (int -> int) * int -> int = <fun>
# let plus5 = fun x -> x+5;;
val plus5 : int -> int = <fun>
# let fois8 = fun x -> x*8;;
val fois8 : int -> int = <fun>
# app(plus5,12);;
- : int = 29
# app(fois8,12);;
- : int = 108
```

Fonctions en tant qu'argument d'un paramètre

```
# let app_couple(f,c,a) =
  let (x1,x2)=c in f (x1+a) * f (x2+a) ;;

val app_couple : (int -> int) * (int * int) * int -> int = <fun>

# app_couple(plus5,(1,1),3);;
- : int = 81

# app_couple((fun y -> y * y),(1,1),3);;
- : int = 256
```

Fonctions en tant que résultats

```
# let decalage(f,a) = fun y -> f (y+a) ;;
val decalage : (int -> int) * int -> int -> int = <fun>

# decalage(plus5,8);;
- : int -> int = <fun>

# decalage(plus5,8) 1 ;;
- : int = 14

# let composition(f,g) = fun y -> f(g(y+1)+1) +1;;
val composition : (int -> int) * (int -> int) -> int -> int = <fun>

# composition(plus5,plus5);;
- : int -> int = <fun>

# composition(plus5,plus5) 1;;
- : int = 14
```

Curryfication

Une fonction définie sur un argument couple **peut se voir** comme une fonction sur un argument dont le résultat est une fonction qui attend un second argument ...

```
# let h1(x,y) = x*y/2;;
val h1 : int * int -> int = <fun>
```

```
# let h2 = fun x -> fun y -> x*y/2;;
val h2 : int -> int -> int = <fun>
```

```
# h1(3,4);;
- : int = 6
# h1 3;;
This expression has type int but is here used with type int * int
# h2 3 4 ;;
- : int = 6
# h2 3 ;;
- : int -> int = <fun>
```

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 succinct.
- 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 OCAML

```
# 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)
```

Définitions de Types

Motivations :

- Pour décrire des nouvelles données structurées.
- Les définitions de types peuvent être polymorphes.
- Le filtrage permet un accès simple aux composants de structures de données complexes.
- Deux grandes familles : les types produit et les types somme.

Les types sans paramètres

Syntaxe générale :

```
# type nouveau_nom = typedef ;
```

Types énumérés

Syntaxe :

```
# type nouveau_nom = Const1 | Const2 | ... | Constn ; ; Exemple :
```

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

```
# Pique;;
- : couleur = Pique
```

```
# Pique = Carreau;;
- : bool = false
```

```
# let est_rouge x = match x with
  Pique -> false
  | Coeur -> true
  | Carreau -> true
  | Trefle -> false;;
```

```
val est_rouge : couleur -> bool = <fun>
```

```
# est_rouge Pique;;
- : bool = false
```

```
# est_rouge Coeur;;
- : bool = true
```

```
# type figure = As | Roi | Dame | Cavalier | Valet |
  Dix | Neuf | Huit | Sept;;
type figure = As | Roi | Dame | Cavalier | Valet |
  Dix | Neuf | Huit | Sept
```

Type avec étiquettes

Syntaxe :

```
# type nouveau_nom = etiquette of type1 * ... * typen ; ;
```

Chaque définition de type avec étiquettes rends disponible des *constructeurs* de ce type qui sont identiques aux étiquettes présentes dans la définition.

Exemple :

```
# type carte_combinee = Carte of figure * couleur;;
```



```

type carte_combinee = Carte of figure * couleur

# Carte(As,Trefle);;
- : carte_combinee = Carte (As, Trefle)
# Carte(As,Trefle) <> Carte(As,Coeur);;
- : bool = true

```

Remarque sur les constructeurs

Ces constructeurs ne sont pas des fonctions : ils doivent être toujours employés avec tous leur paramètres!

Exemple :

```

# type carte_combinee = Carte of figure * couleur;;
type carte_combinee = Carte of figure * couleur
# Carte;;
Characters 0-5:
  Carte;;
  ^^^^^

```

The constructor Carte expects 2 argument(s),
but is here applied to 0 argument(s)

4 façons d'oublier la couleur...

```

# let figure_carte x = match x with
  Carte(a,b) -> a;;
val figure_carte : carte_combinee -> figure = <fun>

# let figure_carte x =
  let Carte(a,b) = x in a;;
val figure_carte : carte_combinee -> figure = <fun>

# let figure_carte (Carte(a,b)) = a;;
val figure_carte : carte_combinee -> figure = <fun>

# let figure_carte (Carte(a,_)) = a;;
val figure_carte : carte_combinee -> figure = <fun>

```

Le jeu du Tarot



Comment représenter les cartes ?

Les cartes de Tarot (Alternative I)

```

# type carte_tarot_1 =
  Excuse | Atout of int | Carte of figure * couleur;;
type carte_tarot_1 = Excuse
  | Atout of int
  | Carte of figure * couleur

```

Les cartes de Tarot (Alternative II)

```

# type carte_tarot_2 =
  Roi of couleur
  | Dame of couleur
  | Cavalier of couleur
  | Valet of couleur
  | Petite_carte of couleur * int
  | Atout of int
  | Excuse ;;

```

Types avec paramètre unique

Syntaxe :

```

# type 'a nouveau_nom = typedef ;;

# type 'a tri_meme_type = TripletM of 'a * 'a * 'a;;
type 'a tri_meme_type = TripletM of 'a * 'a * 'a

# ("bb", "cc", "aa");;
- : string * string * string = "bb", "cc", "aa"

# TripletM("bb", "cc", "aa");;
- : string tri_meme_type = TripletM ("bb", "cc", "aa")

```

Types avec plusieurs paramètres

Syntaxe :

```
# type ('a1, ..., 'an) nouveau_nom = typedef ;

# type ('a, 'b, 'c) tri_diff_type = TripletD of 'a * 'b * 'c;;
type ('a, 'b, 'c) tri_diff_type = TripletD of 'a * 'b * 'c

# TripletD(3,4,5);;
- : (int, int, int) tri_diff_type = TripletD (3, 4, 5)

# TripletD(3,"bonjour",5.4);;
- : (int, string, float) tri_diff_type =
  TripletD (3, "bonjour", 5.4)
```

Types abstraits de données (TAD)

- Manipulation de domaines complexes.
- La manipulation est **indépendante** de la **représentation** particulière du domaine.
- On se donne :
 - Un nom pour **domaine abstrait**
 - Des **opérations de construction** d'une représentation concrète vers le domaine abstrait
 - Des **opérations de test** pour les différents constructions.
 - Des **opérations d'accès** d'un objet abstrait vers les composantes d'une représentation concrète.

Forme générale d'un TAD

Domaine :

dom

Constantes :

$c_1, \dots, c_n : dom$

Opérations de construction :

$op_1 : \{x : d_1 \mid P_1(x)\} \rightarrow dom$

⋮

$op_m : \{x : d_m \mid P_m(x)\} \rightarrow dom$

Opérations de test :

$est_{c_1}, \dots, est_{c_n} : dom \rightarrow bool$

$est_{op_1}, \dots, est_{op_m} : dom \rightarrow bool$

Opérations d'accès :

$acces_1 : \{x : dom \mid est_{op_1}(x)\} \rightarrow \{y : d_1 \mid P_1(y)\}$

⋮

$acces_m : \{x : dom \mid est_{op_m}(x)\} \rightarrow \{y : d_m \mid P_m(y)\}$

Autres opérations :

⋮

Quelques variantes

On peut **omettre** ou **rajouter** quelques items selon les cas :

- Si la propriété $P_i(x)$ est toujours *true*, alors on l'écrit pas.
- Si le domaine est construit à partir d'une seule constante ou d'une seule opération de construction, alors **inutile** d'écrire l'opération de test car elle est équivalente à $est(x) = true$.
- Si le domaine est construit à partir de m opérations de construction, alors $m - 1$ opérations de test suffisent.
- Si un domaine d_i est un produit cartésien $a_1 \times \dots \times a_k$, alors on peut **remplacer** l'opération d'accès

$acces_i : \{x : dom \mid est_{op_i}(x)\} \rightarrow \{y : d_i \mid P_i(y)\}$

par k opérations de la forme

$acces_{i_1} : \{x : dom \mid est_{op_i}(x)\} \rightarrow$
 $\{proj_1(y) \mid y : d_i \text{ et } P_i(y)\}$

⋮

$acces_{i_k} : \{x : dom \mid est_{op_i}(x)\} \rightarrow$
 $\{proj_k(y) \mid y : d_i \text{ et } P_i(y)\}$

Exemple : jour

Domaine :

jour (jours vus comme des entiers de 1 à 31)

Opérations de construction :

cons_jour : $\{x : entier \mid 1 \leq x \leq 31\} \rightarrow jour$

Opérations d'accès :

numéro_jour : $jour \rightarrow \{x : entier \mid 1 \leq x \leq 31\}$

En OCAML

```
# type jour = J of int;;
```

```

type jour = J of int

# let cons_jour x =
  if 1 <= x & x <= 31
  then J(x)
  else failwith "jour invalide";;
val cons_jour : int -> jour = <fun>

# let numero_jour j = match j with J(x) -> x;;
val numero_jour : jour -> int = <fun>

# cons_jour 43;;
Exception: Failure "jour invalide".

# cons_jour 3;;
- : jour = J 3

```

Exemple : exemplaire

Domaine :
 exemplaire
Opérations de construction :
 cons_livre : livre → exemplaire
 cons_video : video → exemplaire
Opérations de test :
 est_livre, est_video : exemplaire → bool
Opérations d'accès :
 ex_livre : {x : exemplaire | est_livre(x)} → livre
 ex_video : {x : exemplaire | est_video(x)} → video

Exemple : argent

Domaine :
 argent
Opérations de construction :
 francs : entier → argent
 euros : entier → argent
Opérations de test :
 est_francs, est_euros : argent → bool
Opérations d'accès :
 val_argent : argent → entier
En OCAML

```

# type argent = F of int | E of int;;
type argent = F of int | E of int

# let est_francs x = match x with
  F(y) -> true
  | _ -> false;;
val est_francs : argent -> bool = <fun>

# let est_euros x = match x with
  E(y) -> true
  | _ -> false;;
val est_euros : argent -> bool = <fun>

# let val_argent x = match x with
  F(y) -> y
  | E(y) -> y;;
val val_argent : argent -> int = <fun>

```

Exemple : Les cartes de couleur

Domaine :
 couleur
Constantes :
 pique, coeur, carreau, trèfle : couleur

Domaine :
 figure
Constantes :
 as, roi, dame, valet, dix, neuf, huit, sept : figure

Domaine :

carte_combinee

Opérations de construction :

cons_carte : couleur × figure → carte_combinee

Opérations d'accès :

figure_carte : carte_combinee → figure

couleur_carte : carte_combinee → couleur

Domaine :

main

Opérations de construction :

cons_main : { $\vec{n} \in \text{carte_combinee}^5 \mid \text{Diff}(\vec{n})$ } → main

Opérations d'accès :

carte¹ : main → carte_combinee

:

carte⁵ : main → carte_combinee

Domaine :

carte_tarot

Constantes :

excuse : carte_tarot

Opérations de construction :

cons_tarot_simple : carte_combinee → carte_tarot

cons_atout : { $n : \text{entier} \mid 1 \leq n \leq 21$ } → carte_tarot

Opérations de test :

est_carte_simple, est_atout : carte_tarot → bool

Opérations d'accès :

couleur_de_tarot : { $x : \text{carte_tarot} \mid \text{est_carte_simple}(x)$ }
→ couleur

figure_de_tarot : { $x : \text{carte_tarot} \mid \text{est_carte_simple}(x)$ }
→ figure

atout_de_tarot : { $x : \text{carte_tarot} \mid \text{est_atout}(x)$ } →
{ $n : \text{entier} \mid 1 \leq n \leq 21$ }

[Le Tarot \(http://www.shogix.net/rules.php\)](http://www.shogix.net/rules.php) en OCAML

```
(* type couleur *)

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

(* Constructeurs pour les constantes de couleur *)
```

```
#let pique = Pique;;
#let coeur = Coeur;;
#let carreau = Carreau;;
#let trefle = Trefle;;

(* type figure *)
#type figure =
  As | Roi | Dame | Cavalier | Valet | Dix | Neuf | Huit | Sept;;

(* Constructeurs pour les constantes de figure *)
#let as =As;;
#let roi =Roi;;
#let dame =Dame;;
#let cavalier=Cavalier;;
#let valet =Valet;;
#let dix =Dix;;
#let neuf =Neuf;;
#let huit =Huit;;
#let sept =Sept;;

(* type carte_combinee *)
#type carte_combinee = Carte of figure * couleur;;

(* opération de construction du type carte *)
#let cons_carte(f,c) = Carte(f,c);;

(* opérations d'accès du type carte *)
#let figure_carte e = match e with Carte(f,c)->f;;
#let couleur_carte e = match e with Carte(f,c)->c;;

(* type carte_tarot *)
#type carte_tarot =
  Excuse | Simple of carte_combinee | Atout of int;;

(* opérations de construction du type carte_tarot *)
#let int_to_carte_tarot n =
  if 0 <= n & n <= 21
  then cons_atout n
  else failwith"numéro d'atout invalide";;
```

```
(* opérations de test du type carte_tarot *)
#let est_carte_simple c =
  match c with Simple(_) -> true | _ -> false;;
#let est_atout c =
  match c with Atout(_) -> true | _ -> false;;

(* opérations d'accès du type carte_tarot *)
#let couleur_de_tarot e =
  match e with
  Simple(Carte(_,c)) -> c
  | _ -> failwith "Pas une carte simple";;

#let figure_de_tarot e =
  match e with
  Simple(Carte(f,_)) -> f
  | _ -> failwith "Pas une carte simple";;

#let atout_de_tarot e =
  match e with
  Atout(x) -> x | _ -> failwith "Pas un atout";;
```

Codage d'autres opérations d'un TAD

- **Abstrait** : on utilise uniquement les opérations abstraites définies dans le TAD.
- **Concret** : on utilise les primitives du langage de programmation associées aux domaines concrets utilisés dans le TAD.

Codage abstrait

```
#let points c =
  if c = excuse
  then 4.5
  else if est_carte_simple c
  then let f = figure_de_tarot c in
    if f = roi
    then 4.5
    else if f = dame
    then 3.5
    else if f = cavalier
```

```
then 2.5
  else if f = valet
  then 1.5
  else 0.5
  else let n = atout_de_tarot c in
    if n=1 or n=21
    then 4.5
    else 0.5;;
```

```
val points : carte_tarot -> float = <fun>
```

Codage concret

```
#let points c = match c with
  Excuse -> 4.5
  | Simple(Carte(f,_)) -> (match f with
    Roi -> 4.5
    | Dame -> 3.5
    | Cavalier -> 2.5
    | Valet -> 1.5
    | _ -> 0.5)
  | Atout(1) -> 4.5
  | Atout(21) -> 4.5
  | _ -> 0.5;;
val points : carte_tarot -> float = <fun>
```

Quelques remarques

On conclut ici la présentation de la partie "pure" du langage OCaml. On a fait l'impasse sur

- les types enregistrements
- les modules et les foncteurs
- la librairie standard
- la compilation

Vous pourrez approfondir plus avant ces aspects importants.

Mais avant de passer à la partie impure, quelques lignes sur les conventions syntaxiques et les erreurs fréquentes.

Comment mettre (ou pas) les parenthèses

OCaml dispose de conventions syntaxiques qu'il est nécessaire de bien connaître pour pouvoir lire et écrire du code sans trop de parenthèses. En voici les principales :

les définitions de fonctions associées à droite

c'est à dire :

```
fun x -> fun y -> fun z -> e
```

est bien la même chose que

```
fun x -> (fun y -> (fun z -> e))
```

l'application de fonctions associée à gauche

c'est à dire :

```
f a b c d
```

est bien la même chose que

```
((((f a) b) c) d)
```

l'application à priorité sur la définition de fonction

c'est à dire :

```
fun x -> x 1
```

veut dire

```
fun x -> (x 1)
```

définition de fonctions à plusieurs paramètres :

si on écrit

```
fun x y z -> x+y+z
```

ceci est une abréviation pour

```
fun x -> fun y -> fun z -> x+y+z
```

filtrage sur les constructeurs :

la convention précédente signifie que la définition suivante

```
fun Carte (f,c) -> f
```

signifie

```
fun Carte -> fun (f,c) -> f
```

ce qui donne, bien évidemment, un erreur

```
# fun Carte (f,c) -> f;;
```

Characters 4-9:

```
fun Carte (f,c) -> f;;  
^^^^^
```

The constructor Carte expects 2 argument(s),
but is here applied to 0 argument(s)

Vous devez écrire plutôt

```
fun (Carte (f,c)) -> f
```

et de même

```
let figure (Carte (f,c)) = f
```

cas particulier de fonction :

si on utilise "function", on sait qu'il y a un seul paramètre, et alors on peut écrire :

```
function Carte (f,c) -> f
```