Introduction à OCAML

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

# 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 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
```

Notation équivalente :

$$e_1 \texttt{ or } e_2 \equiv e_1 \texttt{||} e_2$$

L'opérateur de conjonction

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

Notation équivalente :

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

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 in
```

Structures conditionnelles multiples

FILTRAGE/PATTERN-MATCHING

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

Syntaxe

match e with

$m_1 -> expr_1$

$| m_2 -> expr_2$

$| \vdots \ \ -> $

$| m_n -> expr_n$

$| \_ \ -> expr$

avec $m_1, m_2, \ldots, m_n$ du même type et

$expr_1, expr_2, \ldots, expr_n, expr$ du même type.

est une expression équivalente au conditionnelle simple :

if $e = m_1$

then $expr_1$

else if $e = m_2$

then $expr_2$

$\vdots$

if $e = m_n$ then $expr_n$ 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` $var = expr_1$

```
# 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 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` $var_1 = expr_1 \ldots$ `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
```

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 la liste $par_1, \ldots, par_n$ 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 = "bbbbacccc"
# a_au_milieu "bbbb" 4;;
This expression has type int but is here used with type strin
```


Les fonctions avec nom : cas particulier

Syntaxe `let` $nom\ par_1\ =\ expr$

La fonction nom possède un seul paramètre par_1

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


Les fonctions avec nom : cas particulier

Syntaxe `let` $nom\ (par_1^1, \ldots, par_1^n)\ =\ expr$

- La fonction nom possède un seul paramètre qui est un n-uplet de la forme $(par_1^1, \ldots, par_1^n)$.
- 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 *
# min(3.,5.);;
This expression has type float * float but is here used with
```


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` $par_1 \ldots par_n$ `->` $expr$

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

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 *
# (fun (x,y) -> x*2*y) (2,3);;
- : int = 12
```

Ou alternativement...

Syntaxe `function` par `->` $expr$

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


Cas particulier

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

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 = <f

# 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) + a;;
val decalage : (int -> int) * int -> int -> int = <fun>

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

# decalage(plus5,8) 1 ;;
- : int = 22
```

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

# 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 *
# 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 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 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 enumerés

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* ; ;

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
```



```
# 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>
```


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 :

$c_1, \ldots, c_n \ : dom$

Opérations de construction :

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

$\vdots$

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

Opérations de test :

$est\_c_1, \ldots, est\_c_n \ : dom \to bool$

$est\_op_1, \ldots, est\_op_m \ : dom \to bool$

Opérations d'accès :

$acces_1 \ : \{x : dom \mid est\_op_1(x)\} \to \{y : d_1 \mid P_1(y)\}$

$\vdots$

$acces_m \ : \{x : dom \mid est\_op_m(x)\} \to \{y : d_m \mid P_m(y)\}$

Autres opérations :

$\vdots$

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 $a_1 \times \ldots \times a_k$, alors on peut remplacer l'opération d'accès

$$acces_i \ : \{x : dom \mid est\_op_i(x)\} \to \{y : d_i \mid P_i(y)\}$$

par k opérations de la forme

$$acces_{i_1} \ : \{x : dom \mid \ est\_op_i(x)\} \to \{proj_1(y) \mid y : d_i \text{ et } P_i(y)\}$$

$\vdots$

$$acces_{i_k} \ : \{x : dom \mid \ est\_op_i(x)\} \to \{proj_k(y) \mid y : d_i \text{ et } P_i(y)\}$$

Exemple : `jour`

Domaine :

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

Opérations de construction :

`cons_jour` : $\{x : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\} \to \texttt{jour}$

Opérations d'accès :

`numéro_jour` : `jour` $\to \{\texttt{x} : \texttt{entier} \mid 1 \leq \texttt{x} \leq 31\}$

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 $\rightarrow$ exemplaire

cons_video : video $\rightarrow$ exemplaire

Opérations de test :

est_livre,est_video : exemplaire $\rightarrow$ bool

Opérations d'accès :

ex_livre : $\{x : \text{exemplaire} \mid \text{est\_livre}(x)\} \rightarrow$ livre

ex_video : $\{x : \text{exemplaire} \mid \text{est\_video}(x)\} \rightarrow$ video

Exemple : argent

Domaine :

argent

Opérations de construction :

francs : entier $\rightarrow$ argent

euros : entier $\rightarrow$ argent

Opérations de test :

est_francs,est_euros : argent $\rightarrow$ bool

Opérations d'accès :

val_argent : argent $\rightarrow$ 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,trêfle : couleur

Domaine :

figure

Constantes :

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

Domaine :

carte_combinee

Opérations de construction :

cons_carte : couleur $\times$ figure $\rightarrow$ carte_combinee

Opérations d'accès :

figure_carte : carte_combinee $\rightarrow$ figure

couleur_carte : carte_combinee $\rightarrow$ couleur

Domaine :

main

Opérations de construction :

cons_main : $\{\vec{n} \in \texttt{carte\_combinee}^5 \mid \texttt{Diff}(\vec{\texttt{n}})\} \rightarrow \texttt{main}$

Opérations d'accès :

$\texttt{carte}^1$: main $\rightarrow$ carte_combinee

$\vdots$

$\texttt{carte}^5$: main $\rightarrow$ carte_combinee

Domaine :

carte_tarot

Constantes :

excuse : carte_tarot

Opérations de construction :

cons_tarot_simple : carte_combinee $\rightarrow$ carte_tarot

cons_atout : $\{n$: entier $\mid 1 \leq \texttt{n} \leq 21\} \rightarrow$ carte_tarot

Opérations de test :

est_carte_simple,est_atout : carte_tarot $\rightarrow$ bool

Opérations d'accès :

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

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

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

Le Tarot 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 au 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>
```


Induction sur les formules propositionnelles

Syntaxe du calcul propositionnel (rappel)

Soit $\mathcal{R}$ en ensemble dénombrable de lettres propositionnelles.

Définition : L'ensemble $\mathcal{F}_{prop}$ de formules du calcul propositionnel, est le plus petit ensemble t.q.

- $\mathcal{R} \subseteq \mathcal{F}_{prop}$
- Si $A \in \mathcal{F}_{prop}$, alors $\neg A \in \mathcal{F}_{prop}$.
- Si $A, B \in \mathcal{F}_{prop}$, alors $\vee(A, B)$, $\wedge(A, B)$, $\rightarrow (A, B) \in \mathcal{F}_{prop}$.

Exemple :	$\neg(p)$	$\vee(p, p)$	$\rightarrow (\wedge(p, q), \neg(r))$
Notation simplifiée :	$\neg p$	$p \vee p$	$(p \wedge q) \rightarrow \neg r$

Le calcul propositionnel en OCAML (rappel)

```
# type fprop = L of int | Neg of fprop |
               Ou of fprop * fprop |
               Et of fprop * fprop |
               Impl of fprop * fprop;;

# let f1 = Et(Ou(L(1),L(2)),Neg(L(3)));;
val f1 : fprop = Et (Ou (L 1, L 2), Neg (L 3))
```

Les formules comme des arbres

Les formules constitue un ensemble parametré par deux types `'a` et `'b`, où chaque étiquette d'une feuille est de type `'a` et chaque étiquette d'un nœud interne unaire ou binaire est de type `'b`.

Ceci donne une nouvelle définition du type `('a,'b) AbinII` :

L'ensemble `('a, 'b) AbinII` est le plus petit ensemble t.q.

- Si m est de type `'a`, alors $feuille(m) \in$ `('a, 'b) AbinII`
- Si $c \in$ `('a, 'b) AbinII` et o est de type `'b`, alors $noeud1(o, c) \in$ `('a, 'b) AbinII`.
- Si $c_1, c_2 \in$ `('a, 'b) AbinII` et o est de type `'b`, alors $noeud2(o, c_1, c_2) \in$ `('a, 'b) AbinII`.

Un type abstrait pour `('a, 'b) abinII`

Domaine :

`('a,'b) abinII`

Opérations de construction :

`cons_f` : `'a` → `('a,'b) abinII`

`cons_u` : `'b` × `('a,'b) abinII` → `('a,'b) abinII`

`cons_b` : `'b` × `('a,'b) abinII` × `('a,'b) abinII` → `('a,'b) abinII`

Opérations de test :

`est_f`, `est_u`,`est_b` : `('a,'b) abinII` → `booléen`

Opérations d'accès :

`etiquette_feuille` : `('a,'b) abinII` → `'a`

`etiquette_noeud` : `('a,'b) abinII` → `'b`

`fils_u`, `fils_g`, `fils_d` : `('a,'b) abinII` → `('a,'b) abinII`

Le calcul propositionnel en OCAML (deuxième possibilité)

```
# type ('a,'b) abinII =
  F of 'a |
  N1 of  'b * ('a,'b) abinII |
  N2 of 'b * ('a,'b) abinII * ('a,'b) abinII;;

# type op = Neg | Et | Ou | Impl;;

# let f1 = N2(Et,N2(Ou,F(1),F(2)),N1(Neg,F(3)));;
val f1 : (int, op) abinII =
    N2 (Et, N2 (Ou, F 1, F 2), N1 (Neg, F 3))
```

Fonctions récursives sur les formules

Formules $\subseteq$ Arbres, donc

pour définir une fonction sur les formules on peut utiliser l'un des schémas récursifs sur les arbres.

Schéma récursif pour les formules

$$\begin{aligned} \texttt{f}(a) &= g_1(a, \texttt{f}(\texttt{fils\_u}(a))) \\ &\quad \text{si } \texttt{est\_u}(a) \\ \texttt{f}(a) &= g_2(a, \texttt{f}(\texttt{fils\_g}(a)), \texttt{f}(\texttt{fils\_d}(a))) \\ &\quad \text{si } \texttt{est\_b}(a) \end{aligned}$$

Quelques fois g_1 et g_2 se "ressemblent" beaucoup.

Cas particulier : lors que $\texttt{est\_f}(a)$

Exemple : nb de lettres d'une formule

```
# let rec nblettres f = match f with
      F(_)       -> 1
    | N1(_,a)    -> nblettres a
    | N2(_,a,b) -> nblettres a + nblettres b ;;
```

Exercice : Qui sont les fonctions g_1 et g_2 ?

Preuve de propriétés par induction sur les formules

Exemple : Montrer que toute formule A est équivalente à une formule A' où le connecteur $\neg$ se trouve uniquement à gauche d'une lettre propositionnelle.

Calculs des séquents

Définition : Un séquent est un couple de la forme $\Delta \triangleright \Gamma$, où Δ et Γ sont de multi-ensembles de formules.

Exemple :

$$\begin{array}{lll} p, p, p \rightarrow q & \triangleright & r, p \vee s \\ p \rightarrow q & \triangleright & \\ & \triangleright & p, s \\ & \triangleright & \end{array}$$

Définition d'un calcul des séquents

- On fixe des axiomes (des séquents particulièrs)
- On fixe des règles d'inférence de la forme $\dfrac{\Delta_1 \triangleright \Gamma_1 \ldots \Delta_n \triangleright \Gamma_n}{\Delta \triangleright \Gamma}$

Le système $\mathcal{G}$

Axiome : $\Delta, A \triangleright \Gamma, A$

Règles d'inférence logiques :

$$\frac{\Delta \triangleright \Gamma, A}{\Delta, \neg A \triangleright \Gamma}\ (\neg\, g) \qquad \frac{\Delta, A \triangleright \Gamma}{\Delta \triangleright \Gamma, \neg A}\ (\neg\, d)$$

$$\frac{\Delta \triangleright A, \Gamma \quad \Delta, B \triangleright \Gamma}{\Delta, A \rightarrow B \triangleright \Gamma}\ (\rightarrow\, g) \qquad \frac{\Delta, A \triangleright B, \Gamma}{\Delta \triangleright A \rightarrow B, \Gamma}\ (\rightarrow\, d)$$

$$\frac{\Delta, A, B \triangleright \Gamma}{\Delta, A \wedge B \triangleright \Gamma}\ (\wedge\, g) \qquad \frac{\Delta \triangleright A, \Gamma \quad \Delta \triangleright B, \Gamma}{\Delta \triangleright A \wedge B, \Gamma}\ (\wedge\, d)$$

$$\frac{\Delta, A \triangleright \Gamma \quad \Delta, B \triangleright \Gamma}{\Delta, A \vee B \triangleright \Gamma}\ (\vee g) \qquad \frac{\Delta \triangleright A, B, \Gamma}{\Delta \triangleright A \vee B, \Gamma}\ (\vee d)$$

Dérivation d'un séquent

Définition : La dérivation d'un séquent $\Gamma \triangleright \Delta$ à partir d'un ensemble de séquents Φ, notée $\Phi \vdash \Gamma \triangleright \Delta$, est un arbre fini tel que

- les nœuds sont des séquents
- chaque feuille est soit une formule de Φ, soit un axiome
- si S est le père des $S_1 \dots S_n$, alors S est obtenu par l'application d'une règle d'inférence sur $S_1 \dots S_n$.
- la racine de l'arbre est le séquent$\Gamma \triangleright \Delta$

Preuves et théorèmes

Définition : La preuve d'un séquent $\Gamma \triangleright \Delta$ est une dérivation de $\Gamma \triangleright \Delta$ à partir de l'ensemble vide de séquents. On dit dans ce cas que $\Gamma \triangleright \Delta$ est un théorème.

Premier exemple de dérivation dans $\mathcal{G}$

$$\dfrac{\dfrac{p \triangleright q}{\triangleright \neg p, q}\,(\neg\ d)}{\neg q \triangleright \neg p}\,(\neg\ g)$$

On a une dérivation de $\neg q \triangleright \neg p$ à partir de $p \triangleright q$: $p \triangleright q \vdash \neg q \triangleright \neg p$.

Deuxième exemple de dérivation dans $\mathcal{G}$

$$\dfrac{\dfrac{p \triangleright \neg p}{\triangleright p \rightarrow \neg p}\,(\rightarrow\ d)}{\neg(p \rightarrow \neg p)\triangleright}\,(\neg\ g)$$

On a $p \triangleright \neg p \vdash \neg(p \rightarrow \neg p)\triangleright$

Premier exemple de théorème dans $\mathcal{G}$

Modus ponens

$$\frac{\dfrac{p \triangleright p, q\ (ax) \qquad p, q \triangleright q\ (ax)}{p, p \to q \triangleright q}\ (\to\ g)}{}$$

On a $\vdash p, p \to q \triangleright q$

Deuxième exemple de théorème dans $\mathcal{G}$

Tiers exclu

$$\frac{\dfrac{p \triangleright p\ (ax)}{\triangleright p, \neg p}\ (\neg\ d)}{\triangleright p \vee \neg p}\ (\vee\ d)$$

On a $\vdash \triangleright p \vee \neg p$

Troisième exemple de théorème dans $\mathcal{G}$

Loi de Pierce

$$\frac{\dfrac{\dfrac{p \triangleright q, p\ (ax)}{\triangleright p \to q, p}\ (\to\ d) \qquad p \triangleright p\ (ax)}{(p \to q) \to p \triangleright p}\ (\to\ g)}{\triangleright ((p \to q) \to p) \to p}\ (\to\ d)$$

On a $\vdash \triangleright((p \to q) \to p) \to p$

Comment transformer quelques dérivations dans $\mathcal{G}$

Théorème : (Affaiblissement) Si $\Delta \triangleright \Gamma$ est dérivable dans le système $\mathcal{G}$, alors $\Delta, A \triangleright \Gamma$ et $\Delta \triangleright A, \Gamma$ le sont aussi.

Théorème : (Contraction) Si $\Delta, A, A \triangleright \Gamma$ est dérivable dans le système $\mathcal{G}$, alors $\Delta, A \triangleright \Gamma$ l'est aussi. Si $\Delta \triangleright \Gamma, A, A$ est dérivable dans le système $\mathcal{G}$, alors $\Delta \triangleright \Gamma, A$ l'est aussi.

Conséquence logique (rappel)

Définition : Un multi-ensemble de formules Γ est conséquence logique d'un multi-ensemble de formules Δ, noté $\Delta \models \Gamma$, si toute interprétation qui satisfait toutes les formules de Δ satisfait au moins une formule de Γ.

Exemple :

$$p, p \rightarrow q \models q, r, s$$

Propriétés fondamentales du système $\mathcal{G}$

Théorème :(Correction) Le système $\mathcal{G}$ est correcte, i.e., si $\Delta \triangleright \Gamma$ est un théorème, alors $\Delta \models \Gamma$.

Théorème :(Complétude) Le système $\mathcal{G}$ est complet, i.e., si $\Delta \models \Gamma$, alors $\Delta \triangleright \Gamma$ est un théorème.