Licence STS, semestre 4 — 2009–2010
Langages et Génie Logiciel (Info 223) — 29 janvier 2010
`http://www.lri.fr/~paulin/LGL`

Cours 2 : champs mutables, tableaux, exceptions

1 Enregistrements à champs modifiables

Un champ d'enregistrement précédé du mot-clé `mutable` est modifiable.

```
type t = {prenom : string; mutable age : int; }
```

La syntaxe pour initialiser et accéder des champs modifiables est la même que pour les autres champs.

```
let v = { prenom = "toto"; age = 10 }
v.prenom ^ "␣a␣" ^ (string_of_int v.age) ^ "␣ans";;
- : string = "toto␣a␣10␣ans"
```

La modification de la valeur associée à un champ modifiable se fait avec l'opérateur `<-`.

```
v.age <- 11;;
- : unit = ()
v.age;;
- : int = 11
```

Copie L'opérateur **with** réalise une *copie* des champs, qu'ils soient modifiables ou non.

```
let titi = { toto with prenom = "titi" };;
val titi : t = { prenom = "titi"; age = 11 }
```

Références et champs mutables Les références ne sont qu'un *sucre syntaxique* pour des enregistrements à un seul champ mutable.

```
type 'a ref = { mutable contents : 'a }
let affect r v = r.contents <- v
let deref r = r.contents
```


Il ne faut pas confondre les deux types suivants :

```
type t = { prenom : string; mutable age : int }
type t = { prenom : string; age : int ref }
```

Typage des références Le typage des références en présence de polymorphisme est délicat. Par exemple, les trois instructions suivantes provoqueraient une erreur de segmentation.

```
let addelt = let acc = ref [] in fun x -> acc:=x::!acc
addelt 2; addelt 3.5
```

CAML introduit la notion de variable de type **monomorphe**. Ces variables se distinguent des variables polymorphes par le caractère `_` avant le nom de la variable : `'_a`, `'_b`, `'_c` etc.

```
addelt;;
val addelt : '_a -> unit = <fun>
addelt 2;;
addelt;;
- : int -> unit = <fun>
addelt 3.5;;
This expression has type float but is here used with type int
```

2 Tableaux

Les tableaux peuvent être créés en OCaml avec la syntaxe `[|...|]`.

```
let t = [|4;5;6|];;
val t : int array = [|4; 5; 6|]
```

Le type des tableaux contenant des éléments de type `'a` est noté `'a array`. L'accès à la case `i` d'un tableau `t` s'écrit `t.(i)`. La mise à jour d'une case s'effectue à l'aide de l'opérateur `<-` (comme pour les champs modifiables des enregistrements).

```
t.(0) <- 10;
- : unit = ()
t.(0);;
- : int = 10
```

 L'indexation des cases d'un tableau contenant n éléments se fait de 0 à $n-1$.

Fonctions d'initialisation On peut aussi créer des tableaux en utilisant les fonctions `Array.make`, de type `int -> 'a -> 'a array`, et `Array.init` de type `int -> (int -> 'a) -> 'a array`.

```
let t = Array.make 4 'a';;
val t : char array = [|'a';'a';'a';'a'|]
let t = Array.init 5 (fun i -> 2 * i);;
val t : int array = [|0; 2; 4; 6; 8|]
```

 Un tableau créé par `Array.make n v` contient la même valeur `v` dans toutes ses cases. Attention donc aux éventuels problèmes de partage !

 Comme pour les références, les tableaux sont des structures monomorphes.

```
let t = [| [] [] |];;
val t : '_a list array = [| [] [] |]
```

3 Exceptions

3.1 Motivation

Il est fréquent de manipuler des fonctions partielles :

```
val head : 'a list -> 'a = <fun>
let head l = match l with [] -> ??? | x::l -> x;;
```

Quelle peut être la valeur renvoyée par cette fonction quand `l=[]` ? Une solution possible est de renvoyer une valeur de type `'a option` défini de la manière suivante :

```
type 'a option = None | Some of 'a
```

Mais l'utilisation est alors alourdie par la nécessité de traiter le cas d'erreur de manière explicite et éventuellement en cascade même si le programme n'utilise la fonction que sur son domaine de définition :

```
let head_2 l = match head l with Some x -> Some (x+2) | None -> None
val head_2 : int list -> int option = <fun>
```

Le mécanisme d'exception permet de traiter ce problème de manière élégante, en levant une *exception* dans le cas d'erreur.

```
let head l = match l with [] -> failwith "vide" | x::l -> x;;
val head : 'a list -> 'a = <fun>
```

Ce mécanisme est transparent au typage. Cette fonction peut s'utiliser en ignorant les cas d'erreurs :

```
let head_2 l = head l + 2
val head_2 : int list -> int = <fun>
```

3.2 Définition et utilisation

Une exception est définie à l'aide du mot clé `exception`. À la manière d'un constructeur de type somme, une exception doit commencer par une majuscule et peut attendre des arguments.

```
exception Mon_exception1;;
exception Mon_exception2 of string * int;;
```

Les exceptions sont des valeurs comme les autres : elles ont le type spécial `exn` et peuvent être passées en argument.

```
Mon_exception1;;
- : exn = Mon_exception1
Mon_exception2("toto",4);;
- : exn = Mon_exception2("toto",4)
```

La fonction prédéfinie `raise : exn -> 'a` permet de *lever* une exception afin d'interrompre le cours du programme.

```
raise Mon_exception1; print_int 4;;
Exception: Mon_exception1
```

Une exception peut être *rattrapée* à l'aide de la construction **try/with** qui définit des gestionnaires d'exceptions.

```
exception E of int;;
let f x = if x=10 then raise (E 10) else 10/x
try f 10 with Division_by_zero -> 0 | E x -> x
```

 Les arguments d'une exception ne peuvent pas être polymorphes.

 Attention à l'imbrication des constructions **match with** et **try with**. On peut délimiter les blocs avec des **begin end** ou des parenthèses.

```
match ... with
  | ... ->
      begin
        try ... with
              | ... -> ...
              | ... -> ...
      end
  | ... -> ...
```