Modules

Regrouper du code

- dès qu'on programme un peu plus que factorielle, on a intérêt à constituer des *conglomérats*
- les *enregistrements* peuvent servir à regrouper des *valeurs*

(structures en C)

```
type prof = {   nom :  string;
                cours :  string;
                semestre :  int;
                annonce :  unit -> string   }

type lib_list = {   map :  'a list -> ('a->'b) -> 'b list
                    flatten :  'a list list -> 'a list
                    show :  'a list -> unit  }
```

  - des données et du code
  - tout est côte à côte

    pas de lien entre les champs de l'enregistrement

- aller au-delà: les *modules*

Modules et types de modules

- modules: valeurs **et types**
- modules et types de modules (appelés *signatures*)

```
module M = struct
  type t = int * string
  let f c = (fst x)+1
end
```

module

:

```
module type S = sig
  type t = int * string
  val f :  t -> int
end
```

signature

- noter que "`t` fait le lien" dans `S`
- majuscules des deux côtés (modules et signatures)
- généralement, on définit `S`, puis on fait

  `module M : S = struct...end`
- on accède à `f` dans `M` en notant `M.f`

  `M.N.f` modules à l'intérieur de modules
- intérêt: regrouper, découper, structurer le code

(espaces de nommage)

Cacher: types abstraits

de

```
module M = struct
  type t = int * string
  let f c = (fst x)+1
end
```

:

```
module type S = sig
  type t = int * string
  val f :  t -> int
end
```

à

```
module M = struct
  type t = int * string
  let f c = (fst x)+1
end
```

:

```
module type S = sig
  type t
  val f :  t -> int
end
```

- `t` est un *type abstrait*
- le module ne donne pas accès à l'implémentation de `t`
- on peut aussi cacher une *valeur* en l'omettant dans la signature

Des modules à partir d'autres modules

- modules paramétrés

```
(* "espace ordonné" *)
module type EspaceOrd =
sig
  type t
  val ord :  t->t->bool
end
```

```
module ABR (M:EspaceOrd) =
struct
  type arbre = ...
  let insere v a = ..  M.ord ..
  let supprime v a = ...
  let parcourt f a = ...
  ...
end
```

- un foncteur est un module paramétré (par un module)

```
module F (M: S) = struct
  let f x = x+1
  let g y z = (M.h y) + (M.g z)
end
```

  - `F` est le foncteur qu'on est en train de définir
  - `M` est le paramètre du foncteur
  - `S` est une signature (définie plus haut

    – on pourrait mettre `sig val g : bli val h : bla end`)
- il faut indiquer explicitement le type du module qui est le paramètre

On est à l'étage

- les modules et les signatures sont des constructions *du langage* `struct...end, sig...end`
  - cela est loin d'être toujours le cas ("rien" en C, classes/objets en Java, ...)
- cependant, les modules ne "vivent" pas avec les valeurs
  - pas de `M;;`
  - on ne peut passer de module à une fonction (mais à un foncteur, oui)
  - pas de `fun t -> (struct...end)`
- les modules habitent *"ailleurs"*
  - notamment parce qu'ils contiennent des types
- les modules et les modules paramétrés constituent un petit *calcul de fonctions* 'au-dessus' de Caml

Modules: récapitulatif

- modules et types de modules (signatures)
- type abstrait: on cache la définition du type
- modules paramétrés (foncteurs)

- structurer, organiser, découper le code
- compilation séparée: un fichier définit "automatiquement" un module
  - fichier `.ml` : `struct...end`
  - fichier `.mli` : `sig...end`

- exercice: lire et comprendre le fichier `set.ml` (librairie pour les ensembles de Caml)
  - sur les machines élèves, ouvrir `/usr/lib/ocaml/3.09.2/set.ml`
  - un foncteur pour faire les ensembles à partir d'un type ordonné