Modules,

en vitesse

Des grains différents

- des valeurs
  - `3 : int`
  - `"hop" : string`
  - `<fun> : int -> bool`
- des brochettes de valeurs
  - `(3,"hop") : int * string`
  - `{ l=3 ; s = "hop" } : { l : int ; s : string }`
- franchir un pas: les modules

  des brochettes de valeurs *et de types*

```
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

Modules et signatures

- les modules et les signatures sont des constructions du langage
- comme ils contiennent des types, les modules habitent *"à l'étage du dessus"*
  - pas de `M;;`
  - pas de `fun t -> (struct...end)`
- typage: la signature *(type de module)* donne une *vue* sur le module

dans le module	dans la signature
`type t = int*string`	`type t = int*string` ou `type t` **type abstrait**
`let f = (fst x)+1`	`val f : int * 'a -> int` ou `val f : t -> int` ou ⟨ *rien du tout* ⟩

Piocher les valeurs dans les modules

- le nom d'un module commence par une majuscule
- le point infixe permet d'accéder à une valeur au sein d'un module
  - `let a = M.f 2`
  - c'est ce qu'on fait dans `List.map`

    (chaque librairie est un module)
- on peut mettre des modules à l'intérieur des modules

  ça donne `M.N.f`

Foncteurs

- 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
```

  - sur cet exemple
    - `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*
- rôle des modules: structurer et organiser le code