# Programmation fonctionnelle

ENSIIE

Semestre 4 – 2019/20

# Structures de données

# Bibliothèque pour les dictionnaires (et les ensembles)

On veut créer une bibliothèque pour les dictionnaires en l'implémentant avec des AVL

Difficultés :

- arbres de recherche $\Rightarrow$ usage d'une fonction de comparaison
- ordre variable suivant l'application
  - en particulier égalité entre clefs

# Paramétricité

1re solution :

- Passer la fonction de comparaison en argument de toutes les fonctions
  - lourd
  - source d'erreurs si plusieurs ordres mélangés

2e solution :

- Associer un ordre avec un type
  - code plus court et plus fiable
  - code un peu moins générique

# Module

Comment associer un ordre à un type ?

- En les mettant dans un module

```
type t = mytype
let compare x y = ...
```

Interface :

```
type t
val compare : t -> t -> int
```

# Module

Comment associer un ordre à un type ?

- En les mettant dans un module

```
module MyOrderedType = struct
  type t = mytype
  let compare x y = ...
end
```

Interface :

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

Modules imbriqués

# Paramétricité

On peut définir les dictionnaires pour n'importe quel module de type ordonné

- on paramètrise le module des dictionnaires par un module de type ordonné

Un foncteur est un module qui dépend d'un autre module
$\simeq$ fonction sur les modules

```
module Map (OT : OrderedType) = struct

  type key = OT.t
  type 'v map = Empty
    | Node of 'v map * key * 'v * 'v map

  let add k v d = ...
end
```

# Utilisation d'un foncteur

```
module MyMap = Map(MyOrderedType)

let d = MyMap.create ()
val d : 'a Map(MyOrderedType).map = ...
```

# Ensembles et dictionnaires dans la bibliothèque standard

`Set.Make(OrderedType)` et `Map.Make(OrderedType)`

Interface de `Set.Make` :

```
type elt (* The type of the set elements. *)
type t   (* The type of sets. *)
val empty: t    (* The empty set. *)
val is_empty: t -> bool
val mem: elt -> t -> bool
val add, remove: elt -> t -> t
val union, inter, diff: t -> t -> t
val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
val cardinal: t -> int
val elements: t -> elt list
```

# Exemple d'utilisation : graphes

Graphe = ensemble de sommets + arêtes (orientées)

Association entre sommets et ensemble des voisins

Graphe dont les nœuds sont des `string`

```
module StringSet = Set.Make(String)

module StringMap = Map.Make(String)

type graph = StringSet.t StringMap.t
```

# Utilisation

Graphe vide : `StringMap.empty`

Ajout d'un sommet `v` : `StringMap.add v StringSet.empty`

Ajout d'une arête de `u` vers `v` : `StringMap.find` + `StringSet.add`

Tester si `v` successeur de `u` : `StringMap.find` + `StringSet.mem`

Traitement sur tous les sommets : `StringMap.fold`

Traitement sur toutes les arêtes : `StringMap.fold` + `StringSet.fold`