Programación Declarativa.

Programación modular.

Selene Linares Arévalo.

Facultad de Ciencias, UNAM

2016-2.

Estructuras

La principal motivación para construir módulos es reunir definiciones relacionadas entre sí.

Estructuras

La principal motivación para construir módulos es reunir definiciones relacionadas entre sí. A este conjunto de definiciones le llamamos *estructura* y se introduce con la siguiente sintaxis:

Estructuras

La principal motivación para construir módulos es reunir definiciones relacionadas entre sí. A este conjunto de definiciones le llamamos *estructura* y se introduce con la siguiente sintaxis:

```
module < ModNombre > =
    struct
        definiciones e implementación
    end
```

Estructuras

La principal motivación para construir módulos es reunir definiciones relacionadas entre sí. A este conjunto de definiciones le llamamos *estructura* y se introduce con la siguiente sintaxis:

```
module < ModNombre > =
    struct
       definiciones e implementación
    end
```

La implementación de un módulo puede incluir definiciones de tipos, excepciones, definiciones `let` y declaraciones para abrir otros módulos (`open`).

Estructuras

La principal motivación para construir módulos es reunir definiciones relacionadas entre sí. A este conjunto de definiciones le llamamos *estructura* y se introduce con la siguiente sintaxis:

```
module < ModNombre > =
    struct
        definiciones e implementación
    end
```

La implementación de un módulo puede incluir definiciones de tipos, excepciones, definiciones `let` y declaraciones para abrir otros módulos (`open`). Cuando un símbolo `x` es definido dentro de la implementación de un módulo `M`, se hace referencia a éste (fuera del módulo) utilizando la notación `M.x`.

Signaturas

Las *signaturas* son interfaces para las estructuras.

Signaturas

Las *signaturas* son interfaces para las estructuras. Una signatura especifica a cuáles componentes de una *estructura* se puede acceder desde fuera del módulo y sus tipos.

Signaturas

Las *signaturas* son interfaces para las estructuras. Una signatura especifica a cuáles componentes de una *estructura* se puede acceder desde fuera del módulo y sus tipos. Una signatura se puede utilizar además para ocultar algunos componentes de una estructura o exportar componentes con tipos restringidos.

Signaturas

Las *signaturas* son interfaces para las estructuras. Una signatura especifica a cuáles componentes de una *estructura* se puede acceder desde fuera del módulo y sus tipos. Una signatura se puede utilizar además para ocultar algunos componentes de una estructura o exportar componentes con tipos restringidos.

```
module type < SigNombre > =
    sig
       declaración de la interfaz
    end
```

Signaturas

Las *signaturas* son interfaces para las estructuras. Una signatura especifica a cuáles componentes de una *estructura* se puede acceder desde fuera del módulo y sus tipos. Una signatura se puede utilizar además para ocultar algunos componentes de una estructura o exportar componentes con tipos restringidos.

```
module type < SigNombre > =
    sig
       declaración de la interfaz
    end
```

Toda estructura posee una signatura por default, generada por el sistema de inferencia de tipos, que presenta todas las definiciones declaradas en la estructura junto con sus tipos más generales.

Estructuras y signaturas

Un módulo que implementa una signatura en particular especifica el nombre de esa signatura en su definición.

Estructuras y signaturas

Un módulo que implementa una signatura en particular especifica el nombre de esa signatura en su definición. Por supuesto, la signatura debe ser definida antes de implementar el módulo.

Estructuras y signaturas

Un módulo que implementa una signatura en particular especifica el nombre de esa signatura en su definición. Por supuesto, la signatura debe ser definida antes de implementar el módulo.

```
module < ModNombre > : < SigNombre > =
    struct
        ...
    end
```

Módulos

Ejemplo1

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val mem : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val size: 'a set -> int
  val union: 'a set -> 'a set -> 'a set
  val inter: 'a set -> 'a set -> 'a set
end
```

Módulos

Ejemplo1

Implementación de conjuntos con listas permitiendo elementos duplicados.

```
module Set1 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x xs = x :: xs
```

Módulos

Ejemplo1

Implementación de conjuntos con listas permitiendo elementos duplicados.

```
module Set1 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x xs = x :: xs
  let rec size xs = match xs with
    | [] -> 0
    | h :: t -> size t + (if mem h t then 0 else 1)
  let union l1 l2 = l1 @ l2
  let inter l1 l2 = List.filter (fun h -> mem h l2) l1
end
```

Módulos

Ejemplo1

Implementación de conjuntos con listas sin elementos duplicados.

```
module Set2 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x xs = if mem x xs then xs else x :: xs
```

Módulos

Ejemplo1

Implementación de conjuntos con listas sin elementos duplicados.

```
module Set2 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x xs = if mem x xs then xs else x :: xs
  let size = List.length
  let union l1 l2 =
     List.fold_left
      (fun a x -> if mem x l2 then a else x :: a) l2 l1
  let inter l1 l2 = List.filter (fun h -> mem h l2) l1
end
```

Compartiendo tipos

Considerando el ejemplo, los tipos `M1.t` y `M2.t` son incompatibles.

Compartiendo tipos

Considerando el ejemplo, los tipos `M1.t` y `M2.t` son incompatibles. Sin embargo, nos gustaría indicar que ambos tipos aunque son abstractos son idénticos.

Compartiendo tipos

Considerando el ejemplo, los tipos `M1.t` y `M2.t` son incompatibles. Sin embargo, nos gustaría indicar que ambos tipos aunque son abstractos son idénticos. Para hacer esto utilizaremos la siguiente sintaxis que nos permite declarar igualdad de tipos sobre tipos abstractos como restricción en una signatura:

SName with type t1=t2 and ...

Compartiendo tipos

Utilizando esta restricción de tipos, podemos declrar que los módulos `M1` y `M2` definen idénticos tipos abstractos:

Compartiendo tipos

Utilizando esta restricción de tipos, podemos declrar que los módulos `M1` y `M2` definen idénticos tipos abstractos:

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t
val create : unit -> t
val add : t -> unit end
```

Compartiendo tipos

Utilizando esta restricción de tipos, podemos declrar que los módulos `M1` y `M2` definen idénticos tipos abstractos:

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t
val create : unit -> t
val add : t -> unit end

# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig type t = M.t val get : t -> int end
```

Compartiendo tipos

Utilizando esta restricción de tipos, podemos declrar que los módulos `M1` y `M2` definen idénticos tipos abstractos:

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t
val create : unit -> t
val add : t -> unit end

# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig type t = M.t val get : t -> int end

# let x = M1.create () in M1.add x ; M2.get x ;;
- : int = 1
```

Funtores

Funtores en Ocaml son "funciones"de estructuras en estructuras. Son utilizados para expresar estructuras parametrizadas: una estructura A parametrizada por la estructura B es un funtor F con un parámetro formal B.

Funtores

Funtores en Ocaml son "funciones"de estructuras en estructuras. Son utilizados para expresar estructuras parametrizadas: una estructura A parametrizada por la estructura B es un funtor F con un parámetro formal B. Funtores se definen utilizando la siguiente sintaxis:

functor (Nombre : SigNombre) -> estructura

Funtores

Funtores en Ocaml son "funciones"de estructuras en estructuras. Son utilizados para expresar estructuras parametrizadas: una estructura A parametrizada por la estructura B es un funtor F con un parámetro formal B. Funtores se definen utilizando la siguiente sintaxis:

functor (Nombre : SigNombre) -> estructura

```
# module Couple = functor ( Q : sig type t end ) ->
  struct type couple = Q.t * Q.t end;;
module Couple :
functor(Q : sig type t end) -> sig type couple = Q.t * Q.
```

Funtores

También contamos con azúcar sintáctica para definir y nombrar funtores:

Funtores

También contamos con azúcar sintáctica para definir y nombrar funtores:

module MName (SName : signatura) = estructura

Funtores

También contamos con azúcar sintáctica para definir y nombrar funtores:

module MName (SName : signatura) = estructura

```
# module Couple  ( Q : sig type t end ) =
  struct type couple = Q.t * Q.t end;;
module Couple :
functor (Q : sig type t end) ->
         sig type couple = Q.t * Q.t end;;
```

Funtores

Ejemplos.