



```

    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> 'b) -> 'a t -> unit
end

# Stack.create ();;
- : '_a Stack.t = <abstr>

# StandardStack.create ();;
- : '_a StandardStack.t = <abstr>

# MyStack.create ();;
- : '_a MyStack.t = {MyStack.sp = 0; MyStack.c = [||]}

# module MyStack: STACK =
  struct
    type 'a t = {mutable sp: int; mutable c: 'a array }
    exception Empty
    let create () = { sp = 0; c = [||] }
    let clear (s: 'a t) = s.sp <- 0; s.c <- [||]
    let increase s x = s.c <- Array.append s.c (Array.create 5 x)
    let push x s =
      if s.sp >= Array.length s.c then increase s x;
      s.c.(s.sp) <- x;
      s.sp <- s.sp + 1
    let pop s =
      if s.sp = 0
      then raise Empty
      else (s.sp <- s.sp - 1; s.c.(s.sp))
    let length s = s.sp
    let iter f s = for i = s.sp - 1 downto 0 do f s.c.(i) done
  end;;
                                module MyStack : STACK

# MyStack.create ();;
- : '_a MyStack.t = <abstr>

# let s = StandardStack.create ();;
val s : '_a StandardStack.t = <abstr>

# StandardStack.push 1 s;;
- : unit = ()

# StandardStack.pop s;;
- : int = 1

# MyStack.push 2 s;;
Characters 15-16:
  MyStack.push 2 s;;
                ^
This expression has type int StandardStack.t but is here used with type
int MyStack.t

# module OurStack = MyStack;;
module OurStack :
  sig
    type 'a t = 'a MyStack.t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end

# let s = MyStack.create ();;
val s : '_a MyStack.t = <abstr>

# OurStack.push 1 s;;
- : unit = ()

# module OurStack = (MyStack: STACK);;
module OurStack : STACK

# let s = MyStack.create ();;
val s : '_a MyStack.t = <abstr>

```

```
# OurStack.push 1 s;;
```

*Characters 16-17:*

```
OurStack.push 1 s;;
```

^

*This expression has type 'a MyStack.t but is here used with type int OurStack.t*

```
# module type ORDER =
```

```
sig
```

```
  type t
```

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

```
end;;
```

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

```
# module OrderedIntPair: ORDER =
```

```
struct
```

```
  type t = int * int
```

```
  let compare (x1, x2) (y1, y2) =
```

```
    if x1 < y1 then -1
```

```
    else if x1 > y1 then 1
```

```
    else if x2 < y2 then -1
```

```
    else if x2 > y2 then 1
```

```
    else 0
```

```
end;;
```

```
module OrderedIntPair : ORDER
```

```
# module OrderedPair (O: ORDER): ORDER =
```

```
struct
```

```
  type t = O.t * O.t
```

```
  let compare (x1, x2) (y1, y2) = 0
```

```
end;;
```

```
module OrderedPair : functor (O : ORDER) -> ORDER
```

```
# module OrderedPair =
```

```
functor (O: ORDER) ->
```

```
struct
```

```
  type t = O.t * O.t
```

```
  let compare (x1, x2) (y1, y2) = 0
```

```
end;;
```

```
module OrderedPair :
```

```
functor (O : ORDER) ->
```

```
sig type t = O.t * O.t val compare : 'a * 'b -> 'c * 'd -> int end
```

```
# module OrderedPair =
```

```
functor (O1: ORDER) ->
```

```
functor (O2: ORDER) ->
```

```
struct
```

```
  type t = O1.t * O2.t
```

```
  let compare (x1, x2) (y1, y2) = 0
```

```
end;;
```

```
module OrderedPair :
```

```
functor (O1 : ORDER) ->
```

```
functor (O2 : ORDER) ->
```

```
sig type t = O1.t * O2.t val compare : 'a * 'b -> 'c * 'd -> int end
```

```
# module OrderedPair (O1: ORDER) (O2: ORDER): ORDER =
```

```
struct
```

```
  type t = O1.t * O2.t
```

```
  let compare (x1, x2) (y1, y2) =
```

```
    let a = 2 * O1.compare x1 y1 + O2.compare x2 y2
```

```
    in if a > 0 then 1 else if a < 0 then -1 else 0
```

```
end;;
```

```
module OrderedPair : functor (O1 : ORDER) -> functor (O2 : ORDER) -> ORDER
```

```
# module MyInt =
```

```
struct
```

```
  type t = int
```

```
  let compare (a: int) b = if a = b then 0 else if a < b then -1 else 1
```

```
  let make (x: int) = x
```

```
end;;
```

```
module MyInt :
```

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

```
# module MyIntPair: ORDER = OrderedPair (MyInt) (MyInt);;
```

```
module MyIntPair : ORDER
```

```
# module IntSet = Set.Make (MyInt);;
```

```
module IntSet :
sig
  type elt = MyInt.t
  and t = Set.Make(MyInt).t
  val empty : t
  val is_empty : t -> bool
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val singleton : elt -> t
  val remove : elt -> t -> t
  val union : t -> t -> t
  val inter : t -> t -> t
  val diff : t -> t -> t
  val compare : t -> t -> int
  val equal : t -> t -> bool
  val subset : t -> t -> bool
  val iter : (elt -> unit) -> t -> unit
  val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
  val for_all : (elt -> bool) -> t -> bool
  val exists : (elt -> bool) -> t -> bool
  val filter : (elt -> bool) -> t -> t
  val partition : (elt -> bool) -> t -> t * t
  val cardinal : t -> int
  val elements : t -> elt list
  val min_elt : t -> elt
  val max_elt : t -> elt
  val choose : t -> elt
end

# module IntPairSet = Set.Make (MyIntPair);;
module IntPairSet :
sig
  type elt = MyIntPair.t
  and t = Set.Make(MyIntPair).t
  val empty : t
  val is_empty : t -> bool
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val singleton : elt -> t
  val remove : elt -> t -> t
  val union : t -> t -> t
  val inter : t -> t -> t
  val diff : t -> t -> t
  val compare : t -> t -> int
  val equal : t -> t -> bool
  val subset : t -> t -> bool
  val iter : (elt -> unit) -> t -> unit
  val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
  val for_all : (elt -> bool) -> t -> bool
  val exists : (elt -> bool) -> t -> bool
  val filter : (elt -> bool) -> t -> t
  val partition : (elt -> bool) -> t -> t * t
  val cardinal : t -> int
  val elements : t -> elt list
  val min_elt : t -> elt
  val max_elt : t -> elt
  val choose : t -> elt
end

# IntSet.singleton (MyInt.make 2);;
- : IntSet.t = <abstr>

#
```