1 Modules and Functors

We have already encountered modules; for example, the modules List and String. There are other modules we are interested in. For example, we would like to use the Set and Map modules.

Functors are like functions in that they take arguments and output something. Functors take modules as arguments and return another module. Also like functions, the arguments and the output of functors have to belong to particular types. A module’s type is given by its signature. The signature is the collection of constructs (functions, types, etc.) that are visible to the user. For example, the List module makes functions fold_left, fold_right and map, among many others available to the users.

We will illustrate the idea of modules and functors with the Set module in Ocaml.

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html

module Set : sig .. end

(* Sets over ordered types. This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance. *)

module type OrderedType = sig .. end

(* Input signature of the functor Set.Make. *)

module type S = sig .. end

(* Output signature of the functor Set.Make. *)

module Make : functor (Ord : OrderedType) -> S with type elt = Ord.t

(* Functor building an implementation of the set structure given a totally ordered type. *)

For example, we can use the functor Set.Make to create a module for sets of strings.

module StringSet = Set.Make(String);;

This creates a new module called StringSet and all the functions types, etc. available in this module are now listed.

Exercise 1 Explore the list of functions available here. Write a function which
1. converts a list of strings to a set of strings.

2. uses the `Set.filter` function to return a set whose strings are shorter than some length.

### 1.1 Signatures

(From page 39 of “Ocaml for Scientists”): Module signatures declare the interfaces to modules. Modules which adhere to a given signature must define all of the constructs declared in the signature but are free to define additional constructs. However, only those constructs declared in the signature will be accessible, or *visible*.

Signatures may contain several different kinds of declaration:

- type declarations in the form `type ...`
- exception declarations in the form `exception ...`
- variable and function type declarations in the form `val ...`
- open the namespaces of other signatures using `open` statements.
- replicate the contents of other signatures using `include` statements.
- other signature declarations, to nest signatures.

Signatures are declared using the syntax:

```
module type NAME = sig ... end
```

where the name of the signature (NAME) is conventionally written entirely in capital letters.

Here is an example of module type `OrderedType`

```
module type OrderedType = sig .. end

(* Input signature of the functor Set.Make. *)

type t

(* The type of the set elements. *)

val compare : t -> t -> int

(* A total ordering function over the set elements. This is a two-argument function f such that f e1 e2 is zero if the elements e1 and e2 are equal, f e1 e2 is strictly negative if e1 is smaller than e2, and f e1 e2 is strictly positive if e1 is... *)
```
**greater than e2. Example: a suitable ordering function is the generic structural comparison function compare. *)**

The generic structural comparison function `compare` belongs to the `Pervasives` module (the initially opened module).

```ocaml
# compare 2 0;;
- : int = 1
# compare "a" "b";;
- : int = -1
# List.sort compare ["a";"b";"c"];;
- : string list = ["a"; "b"; "c"]
# List.sort compare ["c";"b";"e";"b";"a"];;
- : string list = ["a"; "b"; "b"; "c"; "e"]
#
```

Typical definitions of `compare` are as follows. For integers, we could write:

```ocaml
let mycompare a b =
  if a > b then 1
  else if a < b then -1
  else 0
```

**Exercise 2** Recall the car type we defined earlier.

```ocaml
type car = { model:string; year:int };;
let mycar = { model="sienna"; year=2006};;
```

Write a `compare` function which orders cars by the year first and then alphabetically by their make.

**Exercise 3** Write a module with type `car` and the `compare` function from the previous exercise. Use this along with the `Set.Make` functor to create a module for car sets.

**Exercise 4** See what happens when you try to make a set of lists.

```ocaml
module StringSet = Set.Make(List);;
```

Why does the functor `Set.Make` fail here? Hint: Examine the type signature of `List`.
## 1.2 Structures

(From pages 39-40 of “Ocaml for Scientists”): Module structures may contain several different kinds of definition which, combined, implement the internals of the module:

- type definitions in the form `type ...`
- exception definitions in the form `exception ...`
- variable and function definitions in the form `let ...`
- open the namespaces of other module structures using `open` statements.
- replicate the contents of other module structures using `include` statements.
- other module signature and structure definitions, to nest modules.

Module structures are defined using the syntax:

```ocaml
module NAME = struct ... end
```

where the name of the structure (NAME) is required to begin with a capital letter.

Functors are modules too. They are defined using the syntax:

```ocaml
module NAME
  (Mod1: ModuleType1)
  (Mod2: ModuleType2)
  ...
  =
  struct ... end
```