Modules

Modular design has been a fundamental programming technique since the 70s. **Modules** are *separate, inter-changeable components* that represent a certain clearly identified part of the program and contain everything needed for the relevant computation.

Here are the key features of a module:

- It *encapsulates* the code and data needed to implement a particular functionality, into a single packaged unit (often its own file) protected from the rest of the program.
- It provides a clear interface that specifies how clients/other modules can access this functionality. Access to the module is restricted to the possibilities offered through the interface.
- It can be interchanged with any module that provides the same interface.

In OCAML, there are two new language constructs that allow us to create modules: *signatures* (interfaces) and *structures* (the actual module implementations). We will examine both of these in a specific example.

Signatures

A **signature**, also known as a **module type**, specifies the interface for a module. It consists of a series of declarations of provided bindings and types. The signature cannot contain any normally executable code.

For an example, suppose we were trying to build a "Fraction" module to work with integer fractions. Let us think of what this module might contain:

- A new type to represent fractions.
- Ways to create a fraction from an integer or from two integers.
- An exception to throw if we attempt to divide by 0.
- Perhaps bindings for the special "fractions/numbers" zero and one.
- Ways to add, subtract, multiply and divide fractions.
- A way to turn a fraction into a string.
- A way to obtain the numerator and denominator of a fraction.
- A way to determine if two fractions are equal.

The important thing here is that as far as the outside world is concerned, they do not need to know exactly how we are going to do these things, they just need to be able to call them.

This is the job of the signature, it tells everyone else of the capabilities we will offer. Here's how this might look like:

```
(* Signatures are typically named using all caps and underscores where needed *)
module type FRACTION = sig

  (* The type for fractions. Notice that it is "abstract".
     We cannot create elements of this type directly. *)
  type frac

  exception DivisionByZero

  (* Some constant values *)
  val zero : frac
  val one : frac

  (* Functions for creating fractions *)
  val frac_of_int : int -> frac
  val frac_of_pair : int -> int -> frac

  (* Functions that use fractions *)
  val add : frac -> frac -> frac
  val sub : frac -> frac -> frac
  val mul : frac -> frac -> frac
  val div : frac -> frac -> frac
  val string_of_frac : frac -> string
  val float_of_frac : frac -> float
  val pair_of_frac : frac -> int * int    (* num/denom *)
end
```

As another example, imagine a "set" class for holding sets of elements of the same type. Then we might have a signature for that module that looks like this:

```
module type SET = sig
  type 'a set     (* It is a container type, it must take a type parameter *)

  (* Functions for creating/adding to sets *)
  val empty : 'a set
  val insert : 'a set -> 'a -> 'a set
  val set_of_list : 'a list -> 'a set

  (* Functions for manipulating sets *)
  val contains : 'a set -> 'a -> bool
  val is_empty : 'a set -> bool
  val size : 'a set -> int
  val union : 'a set -> 'a set -> 'a set
  ....
end
```

Structures

Structures are the standard way to implement a module. They may define many more things than what a corresponding signature suggests, and these things will not be visible to the rest of the program.

Let us see for example how we might implement the structure for fractions. We need to make certain decisions:

- We will represent a fraction as a pair of a numerator and a denominator.
- We will require these the two terms have been reduced (no common terms) and so that the denominator is a positive number. This is called an **invariant**. It is a property that we will ensure all our fraction values have.
- Functions that create fractions will be responsible for establishing the invariant in the first place.
- Functions that use fractions can rely on that invariant for their work, and they maintain it.
- The fact that our type is abstract and not visible to the outside world means we get to control how our fractions are created. We can therefore control that all fractions have the invariant property. For example, if someone was allowed to create a fraction by simply providing the pair (4, 2), then other functions would try to use it and they may assume the invariant, which would be a mistake in this case.

Here's a possible implementation. It requires a function gcd for computing the greatest common divisor between two numbers.

```
(* Modules are upper-cased *)
module Fraction : FRACTION = struct
  type frac = int * int

  exception DivisionByZero

  let gcd a b = ...

  let zero = (0, 1)
  let one = (1, 1)

  let frac_of_int i = (i, 1)
  let frac_of_pair num denom =
    let d = gcd num denom
    in if denom = 0 then raise DivisionByZero
       else if denom > 0
            then (num / d, denom / d)
            else (- num / d, - denom / d)

  let add (n1, d1) (n2, d2) = frac_of_pair (n1 * d2 + n2 * d1, d1 * d2)
  let sub (n1, d1) (n2, d2) = add (n1, d1) (-n2, d2)
  let mul (n1, d1) (n2, d2) = frac_of_pair (n1 * n2, d1 * d2)
  let div (n1, d1) (n2, d2) = frac_of_pair (n1 * d2 + n2 * d1, d1 * d2)

  let string_of_frac (n, d) = string_of_int n ^ "/" ^ string_of_int d
  let float_of_frac (n, d) = float n /. float d
  let pair_of_frac (n, d) = (n, d)
end
```