Principles of Programming 2011 Fall - Practice 9
Signature, Module, Module Functor

ROPAS
Seungjung Lee, Youngseok Lee

2011/11/21

Purposes of this practice:

- Using the signature and `with` expression
- Using the module of the OCaml
- Composing the module with module functor

1. Number Module

   Let's make a module that has definition of the number type and operation function. `NUMBER` signature represent contents that the number module should have.

```
module type NUMBER = sig
    type t
    val zero: t
    val add: t -> t -> t
    val mul: t -> t -> t
    val print: t -> unit
    val make: string -> t
end
```

   Let's make a module `Integer` with type is integer and operation is integer operation. This module follows `NUMBER` signature.

```
module Integer : NUMBER = struct
    type t = int
    let zero = ???
    let add x y = ???
    let mul x y = ???
    let print x = print_int x
    let make s = int_of_string s
end
```

   Let's make a module `FloatingPoint`. It uses a number type as a float. This module also follows `NUMBER` signature.

```
module FloatingPoint : NUMBER = struct
    type t = float
    ...
end
```

2. Integer Vector with 3 dimension

   Let's make a module `IntVector3`. It collects 3 dimensional int vector type and its operation. This module follows `VECTOR` signature.

   A module type `VECTOR` which represents vector is following.

```
module type VECTOR = sig
    type t
    type elemType

    exception InvalidInput

    val make: elemType list -> t
    val add: t -> t -> t
    val mul: t -> elemType -> t
    val dot: t -> t -> elemType
    val print: t -> unit
    val to_list: t -> elemType list
end
```

   `t` is a type of the user defined vector type. `elemType` is a user-defined type of the vector's element. For extension later, we decide the type `t` of the `IntVector3` is `int list`.

```
module IntVector3: VECTOR = struct
    type t = int list
    type elemType = int

    exception InvalidInput
    ...
end
```

   Due to signature `VECTOR` doens't expose concrete type of `elemType`, we can't put an argument into `mul`'s second argument which has a type `elemType`. In this case, we can use `with` expression to expose type `elemType` outside of the module.

```
module IntVector3: VECTOR with type elemType = int = struct
    type t = int list
    type elemType = int

    exception InvalidInput
```

```
    ...
end
```

3. Number Vector with 3 dimension

   Let's make a module `Vector3` which collects 3 dimension number vector's type and functions. This module is a kind of module functor. So this module takes an number module as an input and make a various kind of the modules. The modules have a same functions but number types are different.

   Let's make `Vector3` module to copy the definition of `IntVector3` module and modify.

```
module Vector3 (Number: NUMBER) : VECTOR = struct
```

   Also, we must expose `elemType` outside of the module using `with` expression.

```
module Vector3 (Number: NUMBER) : VECTOR with type elemType = Number.t = struct
```

4. Number Vector with N dimension

   Let's make a functor module `Vector` which collects N dimension number vector's type and operation functions. This module functor takes two arguments. The one is the number module which follows `NUMBER` signature. and the second represents Vector's character, in this case it has only dimension information, and it follows `TRAIT` signature.

```
module type TRAIT =
sig
    val dim: int
end

module VectorN (Number: NUMBER) (Trait: TRAIT)
    : VECTOR with type elemType = Number.t = struct
...
end
```

   Now we can make 5 dimension floating point vector by using module functor and number module.

```
module FloatVector5 = VectorN (FloatingPoint) (struct let dim = 5 end)

let strList = ["1.37"; "2.90"; "3.22"; "33.22";"33.33"]
let numList = List.map (FloatingPoint.make) strList
let a = FloatVector5.make numList

...
```