COMP 212 Spring 2015
Lab 9

1 Prescriptive Signatures

Consider the following signature for counters:

```
signature COUNTER =
sig

    (* invariant: counter always represents a natural number *)
    type counter

    val zero : counter
    val increment : counter -> counter
    val value : counter -> int

end
```

This says that you can make a zero counter, increment a counter, and ask for the current value of a counter. The invariant is that a counter value should always be a natural number.

Here is an implementation of this signautre:

```
structure TMICounter : COUNTER =
struct

    (* representation invariant: C x always satisfies x >= 0 *)
    type counter = int

    val zero = 0
    fun increment x = x + 1
    fun value x = x

end
```

Task 1.1 What does the following return?

```
- TMICounter.value (TMICounter.increment ~2);
```

Why are you able to get a negative value from a counter function, breaking the above invariant?

Task 1.2 Define a module `IntCounter` that uses a `datatype` to make the `counter` type abstract.

Task 1.3 Run

```
- IntCounter.value (IntCounter.increment ~2);
```

What happens?

2 Substructures and Functors

Here is a signature for a better counter, which provides an additional operation. It includes a `COUNTER` as a substructure, and also provides an operation for incrementing a counter many times.

```
signature BETTER_COUNTER =
sig

    structure C : COUNTER

    (* assuming n >= 0, increment c that many times *)
    val increment_many_times : C.counter * int -> C.counter

end
```

Task 2.1 Define a functor

```
functor BetterCounter(C : COUNTER) : BETTER_COUNTER
```

that makes a better counter from any counter.

3 Descriptive Signatures

Here is a signature describing a type that comes equipped with a comparison function:

```
signature ORDERED =
sig

    type t
    val compare : t * t -> order

end
```

We call such a type an *ordered type*.

For example, here is an implementation giving the usual $\leq$ ordering on integers:

```
structure IntLt : ORDERED =
struct

    type t = int
    val compare = Int.compare

end
```

Here is an implementation that orders (year,month,day) pairs lexicographically by first the year, then the month, then the day.

```
structure YMDOrder : ORDERED =
struct

    type t = int * (int * int)

    fun compare ((y,(m,d)),(y',(m',d')))  =
        case Int.compare (y,y') of
            LESS => LESS
          | GREATER => GREATER
          | EQUAL => (case Int.compare (m,m') of
                          LESS => LESS
                        | GREATER => GREATER
                        | EQUAL => Int.compare (d,d'))


end
```

For example

```
val LESS = YMDOrder.compare ((1999,(1,1)), (1999,(1,2)))
val LESS = YMDOrder.compare ((1999,(1,3)), (1999,(2,2)))
val GREATER = YMDOrder.compare ((2000,(1,1)), (1999,(2,2)))
```

Task 3.1 A better way to write this code is to write a functor that takes two ordered types and orders the pair type lexicographically, and then to apply this functor twice.

Write a functor

```
signature TWO_ORDERS =
sig
    structure O1 : ORDERED
    structure O2 : ORDERED
end

functor PairOrder(T : TWO_ORDERS) : ORDERED
```

such that the result of `PairOrder(T)` orders the type `T.O1.t * T.O2.t` lexicographically (first look at the first component, then use the second component to break ties).

Task 3.2 Apply the functor `PairOrder` to order a pair of integers, each by the usual $\leq$.

Task 3.3 Apply the functor `PairOrder` again to make a module that is equivalent to `YMDOrder`.