Assignment 9

Lazy Lists Revisited

You have seen how to implement a lazy list in ML. In Haskell, a lazy list requires no special effort, since *all* data structures are lazy by default. In particular, the built-in list type is lazy.

1. Define in Haskell an infinite list called *code* that is simply a never-ending sequence of ones: $1, 1, 1, 1, 1, \ldots$;
2. Write a Haskell function *intList* n that will create a sequence of integers from n to infinity: $n, n+1, n+2, \ldots$ (You may **not** use the special built-in list syntax for this; build the list using only the cons operator *(:)*)
3. Write a Haskell function *takeN* that returns the first n elements from a list. (Do not use any standard functions for this.) For example,

   ```
   takeN 4 (intList 10)
   ```

   should evaluate to:

   ```
   [10, 11, 12, 13]
   ```

Stream Equations

1. Define in Haskell the list of all even positive integers and the list of all odd positive integers.

   ```
   evens :: [Int]
   evens =
   ```

   and

   ```
   odds :: [Int]
   odds =
   ```

2. Define a merge function in Haskell that takes two ordered lists and returns the resulting merged list, in order. For instance,

   ```
   merge [1,2,3] [4,5,6]
   ```

   should return

   ```
   [1,4,2,5,3,6]
   ```

```
merge :: [Int] -> [Int] -> [Int]
```

Does the call

```
head (merge evens odds)
```

terminate? Explain why or why not in a few sentences. What about

```
length (merge evens odds)
```

3. Write each of the sequences below as one or more Haskell streams (infinite lists). You may use the *merge* function defined above.

   (a) $0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, \ldots$

   (b) $1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, \ldots$

   (c) $0, 0, 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, \ldots$

   (d) The negative numbers

   For example, the sequence consisting of all zeros can be described as:

   ```
   zeroes :: [Int]
   zeroes = 0 : zeroes
   ```

   Alternatively, a list can be described using a **list comprehension**:

   ```
   [n + 1 | n <- [1,2,3]]
   ```

   evaluates to

   ```
   [2,3,4]
   ```

Simulate effects in a pure functional language

The following two exercises are meant to show how Haskell simulates effects. A monad is simply a pattern that guides you in writing programs in a pure style.

- **Exceptions**

  1. Consider the following ML datatype:

     ```
     datatype  Term = NUM of int | DIVIDE of Term  * Term
     ```

     Define a function in ML

     ```
     interp : Term -> int
     ```

which raises the exception DivZero when dividing by zero. For example, the program

```
interp (DIVIDE (NUM 3, NUM 0)) handle DivZero => 0
```

will return 0

2. Suppose you do not have the exception mechanism available to raise and catch errors, so you have to come up with a way to simulate exceptions. Rewrite the code of your interpreter in such a way that when you try to divide by zero it still signals an error. Note that you need to define a new datatype, say `Result`, and your new interpreter has now the type

   ```
   Term -> Result
   ```

   Rewrite the following invocation of the interpreter:

   ```
   interp (DIVIDE (NUM 3, NUM 0)) handle DivZero => 0
   ```

- **State**

  1. Suppose you want to write an interpreter that counts the number of division operations in your program. You can do it using the imperative features that ML offers:

     ```
     - val count = ref 0;
     - fun interp_state (NUM x ) = x
       |   interp_state (DIVIDE (t1, t2)) = let
                                              val _=(count := !count + 1)
                                             in (interp_state t1) div (interp_state t2)
                                           end
     ```

     Suppose now you want to write the above in Haskell which doesn't offer the assignment feature. You need to simulate what assignment does in a functional manner. Rewrite `interp_state` without using the assignment, in such a way that when you call your new interpreter with the program `DIVIDE(DIVIDE(NUM 8,NUM 1),DIVIDE(NUM 8,NUM 1))` it returns 1 together with the number of divisions which is 3.