433-431 (433-631) — Quiz 1, Haskell programming
Semester 1, 2008

Introduction

The questions in this quiz cover the material on Haskell programming. These are the kinds of questions you can expect on the end-of-semester exam.

Question 1

Suppose that we represent a set of values using a list:

```
type Set a = [a]
```

Part a. Write a function which will eliminate any duplicate values which appear in a list, assuming equality is defined by the `==` function.

```
removeDuplicates :: Eq a => [a] -> Set a
```

Part b. Write a function which takes a set as input, and returns the *power set* of that set as output:

```
powerSet :: Set a -> Set (Set a)
```

The definition of a power set is as follows. Let S be some set. The power set of S is the set of all subsets of S. Remember that S and the empty set are both subsets of S.

Part c. Compared to the type synonym above, why might it be preferable to represent the set type using `newtype`?

```
newtype Set a = Set [a]
```

Question 2

Recall that the Haskell type `IO t` represents a computation which yields a value of type `t`, and may perform side-effects.

Part a. Identify and fix the error in the following piece of code:

```
helloWorld :: IO ()
helloWorld = do
   putStr "Enter your name: "
   input <- getLine
   response <- "Hello " ++ input
   putStr response
```

Part b. What problems might be caused by the use of a function with type '`runIO :: IO a -> a`' (supposing that it existed, and was able to run an `IO` computation and return its value)?

Part c. We could implement the `IO` monad as a kind of state monad like so:

```
data World = World
type IOError = String
newtype IO a = IO (World -> Either IOError (World, a))

instance Monad IO where
   return x = IO (\world -> Right (world, a))
   (IO f) >>= next = do
      IO (\world1 -> case f world1 of
                        Left e -> Left e
                        Right (world2, val) -> case next val of
                                                  IO g -> g world2)
```

The state type `World` is just a trivial token which is threaded through all the `IO` functions. It is supposed to represent the state of the "real world" in which the computation is running (the world we live in). The contents of the token is not important. And it would be ridiculous to think that we could fit the whole world in there anyway. All that matters is the data dependency created by the threading, which ensures that all the side-effects of an `IO` computation are executed in the desired order. We would also need to implement primitive `IO` functions, such as `putStr`, but for the sake of this question, let's assume that can be done in some compiler specific fashion. Note that `IO` computations can fail with an `IOError`, and this is encoded using the `Either IOError (Word, a)` type. For simplicity the `IOError` type is encoded as a string to carry an error message, such as `"file could not be opened"`.

Implement the `catch` function from the Prelude, for the `IO` type defined above:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

You can use the definition of `>>=` for inspiration.

Question 4

Here is the parser monad we built in lectures:

```
newtype Parser a = P (String -> Maybe (String, a))

instance Monad Parser where
   return x = P (\s -> Just (s, x))
   (P p1) >>= next =
      P (\s1 -> case p1 s1 of
                   Nothing -> Nothing
                   Just (s2, val) -> case next val of
                                        P p2 -> p2 s2)
```

Write a parser combinator with this type:

```
parseN :: Parser a -> Int -> Parser [a]
```

Given some parser `p`, and some non-negative integer `n` as arguments, the expression:

```
parseN p n
```

should attempt to parse `n` instances of `p`. It should succeed only if the parser `p` can be applied successfully *at least* `n` times. It should not apply `p` any more than `n` times.

Question 5

Here is the `Either a b` type from the Prelude:

```
data Either a b = Left a | Right b
```

and here is the Scott encoding of the type:

```
left  = \x -> \l -> \r -> l x
right = \y -> \l -> \r -> r y
```

The type constructor '`Either a`' (yes, the `b` is intentionally missing) can be made an instance of the `Monad` class like so:

```
instance Monad (Either a) where
   return x           = Right x
   (Left x) >>= next  = Left x
   (Right y) >>= next = next y
```

The idea is that the `Left` variant represents an "error", which is propagated, and the `Right` variant represents "success".

Rewrite the definition of `>>=` from above using the Scott encoding of the type. It might help to start from this version which does not use pattern matching:

```
bind e next = case e of
                 Left x -> Left x
                 Right y -> next y
```

Question 6

(As seen on the discussion board, and implemented in terpie).

In an early lecture, Paul asked a good question about using `catch` from the Prelude to handle any exceptions that might arise when you try to read a file. The problem is that `catch` has this type:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

And `readFile` has this type:

```
readFile :: FilePath -> IO String
```

You run into trouble if you use it like this:

```
catch readFile handler
```

where handler is your exception handler function (not defined here). The problem is that the types seem to be forcing you to return the same type as `readFile`, but that only makes sense if the file could actually be opened. If the file can't be opened then we have no `String` to return!

The solution is to massage the output type of `readFile` into something that accommodates errors, such as the `Maybe a` type, or the `Either String b` type (where the `String` is the error message). Write a function which tries to open a file, and if it fails, returns an error:

```
-- type synonyms to help with readability
type Error = String
type FileContents = String

safeReadFile :: FilePath -> IO (Either Error FileContents)
```

You should use the Prelude `catch` as part of your answer. You can turn an `IOError` into a string using `show`.