Name: ______________________________

COSC 8, Winter 2008

Final Exam

March 10, 2008

Problem	1	2	3	4	5	6	**Total**
Grade							
Points	20	15	15	15	20	15	**100**
Grader							

General Instructions

There are six (6) problems on this exam. Please check now that you have a complete exam booklet with eleven (11) numbered pages. Please write your name on *each page* of the exam.

Be sure to look at *all* of the problems. Some are more difficult than others. Don't waste a lot of time on a problem that is giving you a hard time—if you get stuck, move on to another problem, and then return to it later.

There is space provided to answer each question, and you may request extra paper if you so desire. If you do use extra paper to record your solutions, be sure to write your name prominently on each extra page, and clearly indicate which problem is being answered there. You do not need to turn in any extra scratch-paper you use while working out your solutions; only the pages with your answers on them need to be submitted.

To help insure that you don't accidentally miss any of the questions, each section where an answer is requested has been marked with a $\Longleftarrow$ symbol in the right margin.

Please write neatly—the course staff reserves the right to ignore illegible answers. When writing solutions that involve program code, proper indentation of your code is important!

This examination covers material through Lecture 27.

You may not consult any books, notes, study-guides, or other outside material during the exam period, except for one sheet of 8.5 by 11 inch notebook paper which you are allowed to make notes on (both sides). The notes may be hand-written or typed in 10 point or bigger font. You will also be supplied with a mini-manual of list functions.

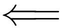
1. (20 Points) **Streams**.

   (a) (8 Points) In class we saw how to construct a stream of ones and a stream of Fibonacci numbers recursively in a way that did not require defining a new function or a different stream. For example,

   ```
   ones = 1 : ones
   ```

   Show how to generate the stream of non-negative powers of 2. That is, define `pow2` to be the stream:

   ```
   [1,2,4,8,16,32, ...]
   ```
   .

   (Of course, the ... is not valid Haskell.) You are allowed to use only arithmetic operators (sections are allowed) and the higher-ordered functions that are defined for lists. You are not allowed to use any stream except `pow2`. (That is, you can't use `ones`, `naturals`, `[1..]` etc.) 

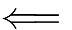
   (b) (12 Points) In class it was asked if we could list everything in an infinite stream of infinite streams. The concern was that if you tried to list the first stream you would never finish. I said that it was possible to make a list where every item in the infinite number of infinite lists appears eventually. The trick is to treat each stream as a row in an infinite matrix, and then process the elements by diagonals. We can use the following order, where the ordered pair (i, j) represents the thing in row i and column j:

   $(1, 1)$
   $(1, 2), (2, 1)$
   $(1, 3), (2, 2), (3, 1)$
   $(1, 4), (2, 3), (3, 2), (4, 1)$

   Note that things in the first row sum to 2, in the second row sum to 3, in the third row sum to 4, etc. Eventually every ordered pair of positive integers will appear in the list.

   Create this stream that includes all possible ordered pairs:

   ```
   allPairs = [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3), ...]
   ```

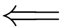
   Unlike the last part, there are no restrictions on what you can do to create this stream. 

2. (15 Points) **Databases**.

Consider the following two tables:

```
ArtifactID       Material       Table name: material    Key: "ArtifactID"
Dewing101         Silver
Dewing1009        Silver
Boston21.1219     Amethyst
Boston01.7545     Rock Crystal
Aegina E1         Marble
Boston03.771      Silver
```

```
ArtifactID       Type          Table name: type    Key: None
Aegina E1        Sculpture
Boston01.7545    Gem
Boston03.771     Sculpture
Boston21.1219    Gem
Dewing1009       Coin
Dewing101        Coin
```

We want to have a single table with columns ArtifactID, Material, and Type (in that order). It should contain only rows where the Material is Silver or the Type is Sculpture. Write database commands that will create such a table from the given tables. (The commands should depend on the table and column names, but not the specific contents. For these tables there should be four rows in the final table.) 

3. (15 Points) **Parsing**. The following is part of the BNF for a simple programming language:

```
statement  = while  | if | assignment | block
block      = '{' statement ';'  [statement ';']... '}'
if         = "if" predicate "then" statement "else" statement
while      = "while" predicate "do" statement
assignment = variable '=' expression
predicate  = ...
variable   = ...
expression = ...
...
```

We define the data type:

```
data Statement = Block [Statement]
               | While Pred Statement
               | If Pred Statement Statement
               | Assignment Var Expr
     deriving Show
```

and we also define data types for `Pred`, `Var`, and `Expr`, although these definitions not given here.

You will be asked to write parsers for `block` and `if`, assuming that the following parsers exist:

```
statement  :: Parser Statement
while      :: Parser Statement
assignment :: Parser Statement
predicate  :: Parser Pred
variable   :: Parser Var
expression :: Parser Expr
```

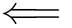
(continued next page)

Assume that these other parsers deal with spaces in the input string. You should assume that ParserX has been imported. Remember that the declaration of this module exports the following items:

```
module ParserX(Parser,
          (#), (-#), (#-), (#:), (#++), (!), (>->), (?), lit, letter, space,
          char, takeParse, takeWhileParse, cons, returnParse, failParse,
          takeRepeatParse, parseSpaces, parseWord, wordChoice, elimSpacesAround,
          module Data.Char)
```

For your convenience we repeat the BNF here:

```
statement  = while  | if | assignment | block
block      = '{' statement ';'  [statement ';']... '}'
if         = "if" predicate "then" statement "else" statement
while      = "while" predicate "do" statement
assignment = var '=' expression
predicate  = ...
variable   = ...
expression = ...
...
```

Write the parsers block and if. 

4. (15 Points) **Monads**. We saw that making `Parser` an instance of `Monad` and `MonadPlus` simplified writing our Parser module by giving us easier ways to write functions `#`, `>->`, `!`, `?`, etc. However, it is possible to write parsers using only the operations defined in `Monad` and `MonadPlus`:

```
instance Monad Parser where
  return value                  -- equivalent to returnParse in Parser
    = Parser (\str -> Just (value, str))
  fail string                   -- equivalent to failParse in Parser
    = Parser (\str -> Nothing)
  parser >>= function
    = Parser $ \ firstString ->
        case runParser parser firstString of
          Nothing -> Nothing
          Just (intermediateValue, intermediateString) ->
            runParser (function intermediateValue) intermediateString


instance MonadPlus Parser where
  mzero       = Parser (\str -> Nothing)
  x `mplus` y = Parser (\str -> runParser x str `mplus` runParser y str)
```

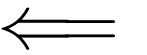
Consider the following parser for a SimpleTree:

```
data SimpleTree  = Leaf
                 | Branch SimpleTree SimpleTree
  deriving Show

simpleTree :: Parser SimpleTree
simpleTree = simpleLeaf ! simpleBranch

simpleLeaf :: Parser SimpleTree
simpleLeaf = lit '*' >-> (\_ -> Leaf)

simpleBranch :: Parser SimpleTree
simpleBranch
    = (lit '('  -# simpleTree # simpleTree #- lit ')')
      >-> (\(left, right) -> Branch left right)
```

Re-write the parsers `simpleTree`, `simpleLeaf`, and `simpleBranch` using monadic operations or `do` statements. You are allowed to use the `lit` parser, but not the `#`, `-#`, `#-`, `>->`, `!`, or `?` functions. 

5. (20 Points) **Short Answer**.

(a) (4 pts) When studying streams, we noted that the `foldr` function generally cannot be applied to a stream, because it would have to do an infinite number of calculations before getting a value. That is certainly true for

```
sum' numStream = foldr (+) 0 numStream
```

where `numSteam` is an infinite stream of numbers. However, it is not true for

```
concat listStream = foldr (++) [] listStream
```

where `listStream` is an infinite stream of finite lists. The call `take 10 (concat listStream)` will correctly return the first 10 elements of the infinite list. Explain why `concat` works but `sum'` does not. Your explanation should involve facts about the definitions of `foldr` and `++`. ⟸

(b) (4 pts) Backtracking gets its name because the way that it is typically implemented is: whenever there is a choice to be made, choose the first possibility and recursively search all possibilities resulting from that choice, then come back and choose the second possibility and recursively search all possibilities resulting from that choice, then come back and choose the third possibility, etc. until all choices are explored. The computation moves forward until it gets to a solution or dead end, and then "backtracks" to make a different choice at the most recent decision point that still has possibilities remaining.

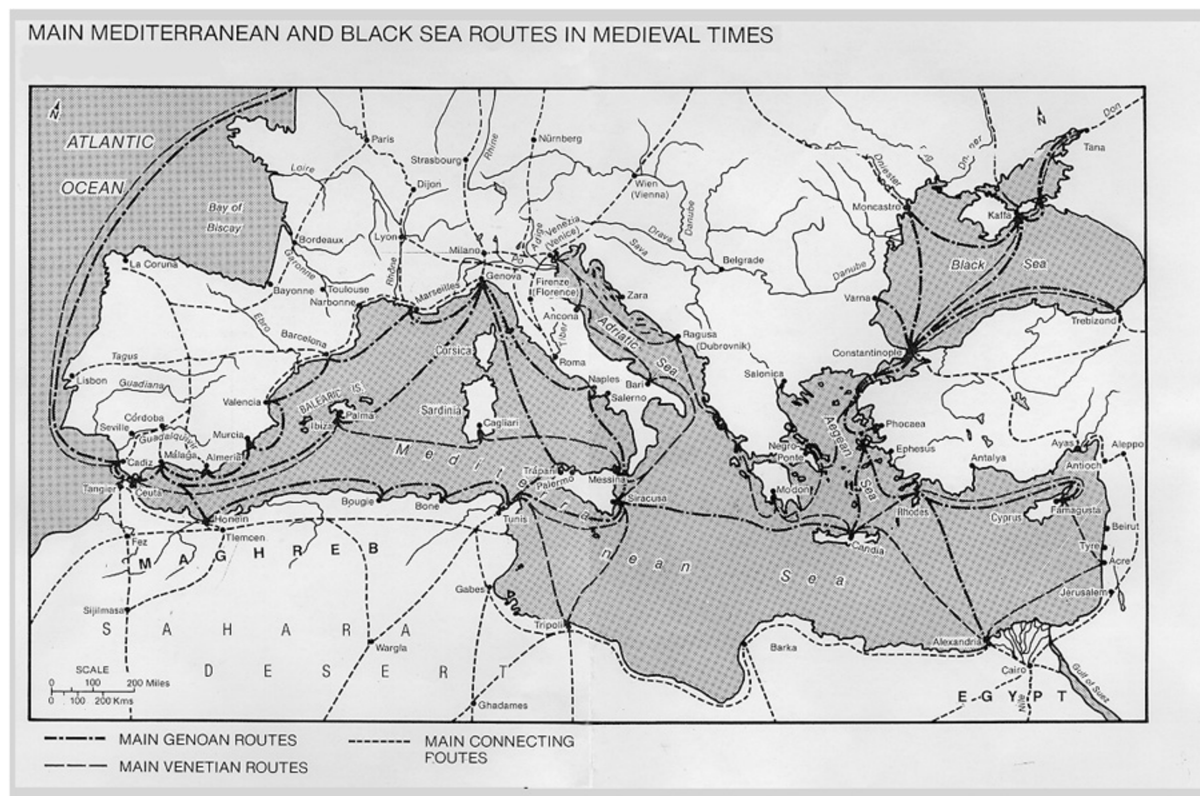
However, backtracking programs we looked at in Haskell used a different approach. Describe this approach. ⟸

(c) (4 pts) The `Region` and `RegionEx` modules have methods `containsS` and `containsR` which determine if a point lies within a `Shape` or `Region`. The natural way to deal with regions that are translated, rotated, or scaled is to transform the region as specified and then test if the point is contained in the transformed region. What do the `Region` and `RegionEx` modules do instead, and why is this computationally less expensive? ⟸

(d) (8 pts) Medieval Mediterranean Trade Routes

Below is a map of trade routes on the Mediterranean during Medieval times. This network of trade routes forms a graph on these medieval cities. For this problem, a subgraph of this map is represented in the graph below.

An edge between two cities indicates that they were trading partners. The cities could trade using a Genoan, Venetian, or Connecting trading route. For example, the entry for Messina means that Messina traded with Cagilari via a Venetian trade route,with Naples via a Genoan trade route, and with Syracuse via a Genoan trade route.



Adjacency List

Messina [(Cagliari, Venetian),(Naples, Genoan), (Syracuse, Genoan)]
Cagliari[(Palma, Venetian), (Messina, Venetian)]
Genova [(Palma, Genoan), (Palermo, Genoan), (Naples, Genoan)]
Naples [(Genova, Genoan), (Messina, Genoan)]
Palma [(Genova, Genoan), (Cagliari, Venetian)]
Syracuse [(Messina, Genoan), (Tunis, Venetian)]
Palermo [(Genova, Genoan), (Tunis, Genoan)]
Tunis [(Syracuse, Venetian), (Palermo, Genoan)]

i) Using the graph described by the adjacency list structure above, construct a BFS tree with Messina as the root. (You may either draw it with directed edges or give the adjacency list structure.) Be sure to process cities in the same order as they appear in the adjacency list. For example, when processing Palma, Genova appears before Cagliari, therefore one would process Genova before Cagliari. ⟸

ii) If we use the BFS tree to determine shortest routes (where length is the number of intermediate cities), what would be the shortest route from Palermo to Messina? ⟸

6. (15 Points) **Tree Walking**. In PS 6 we asked you to parse HTML, and then to do various operations on the tree-like structure that resulted. The types used in that data structure were:

```
type Tag  = String
type Attr = (String,String)
data Node = Element Tag [Attr] [Node]    -- Produced by a tag
          | Text String                  -- Text to be displayed
            deriving (Show, Eq)
```

In this problem we want you to write a function `extractTags` that returns a list of all of the tags in a parsed HTML document represented by this data structure. Its type signature is given below.

`extractTags :: Node -> [Tag]`