

Name: \_\_\_\_\_

COSC 8, Fall 2007

**Final Exam**

December 7, 2007

<b>Problem</b>	1	2	3	4	5	<b>Total</b>
<b>Grade</b>						
<b>Points</b>	20	20	15	20	25	<b>100</b>
<b>Grader</b>						

## General Instructions

There are five (5) problems on this exam. Please check now that you have a complete exam booklet with ten (10) 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  $\Leftarrow$  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 28.

<p>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.</p>
--

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

In class we saw how to construct a stream of ones, the natural numbers, and the Fibonacci numbers recursively in a way that did not require defining a new function. For example,

`ones = 1 : ones`

- (a) (10 Points) Show how to generate the factorials the same way. That is, define `factorials` to be the stream:

`[0!, 1!, 2!, 3!, 4!, ...] = [1, 1, 2, 6, 24, ...]`.

(Of course, the ... is not valid Haskell.) You are allowed to use arithmetic sequence notation (e.g. `[0..]`), arithmetic operators, and the higher-ordered functions that are defined for lists. ⇐

- (b) (10 Points) Using only `ones`, arithmetic operators, and higher-order list functions that work for streams create two mutually recursive streams `evens = [0, 2..]` and `odds = [1, 3..]`. By mutually recursive I mean that `evens` (but not `odds`) can appear in the definition of `odds` and `odds` (but not `evens`) can appear in the definition of `evens`. ⇐

2. (20 Points) **Databases.**

Consider the following three tables:

name	hometown	Table name: people.	Key: "name"
Tom	Hanover		
Lisa	Hanover		
Chris	Miami		
Jim	Portland		

city	recreation	Table name: places.	Key: none
Aspen	skiing		
Hanover	biking		
Hanover	skiing		
Las Vegas	gambling		
Miami	sunbathing		
Miami	vice		

activity	cost	Table name: activities.	Key: "activity"
skiing	expensive		
biking	moderate		
gambling	expensive		
sunbathing	cheap		

(a) Show the result of: `join "city" "hometown" places people`



(b) Show the result of: `innerJoin "recreation" places activities`



3. (15 Points) **Monads.** We saw the Maybe monad in class. It was defined:

```
instance Monad Maybe where
  Just x >>= k = k x
  Nothing >>= k = Nothing
  return      = Just
  fail s      = Nothing
```

Consider the two functions:

```
sqrtM  :: Float -> Maybe Float
sqrtM x = if x >= 0 then return (sqrt x) else fail "sqrt of negative"
```

```
recipM  :: Float -> Maybe Float
recipM x = if x /= 0 then return (1/x) else fail "division by 0"
```

(a) (5 Points) Evaluate the expression, showing the steps in the evaluation process:

```
sqrtM 4 >>= recipM
```



(b) (5 Points) Evaluate the expression, showing the steps in the evaluation process:

```
sqrtM (-4) >>= recipM
```



- (c) (5 Points) A drawback to the Maybe monad is that it lets you know that a failure occurred, but there is no way to pass a message about why. In this part we will modify Maybe to so that Nothing takes an error message as its argument. After this change, recipM 0 will return Nothing "division by 0".

```
data Maybe a = Just a | Nothing String
  deriving Show
```

Modify the definition of the Maybe monad:

```
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return  = Just
  fail s  = Nothing
```

so that it will return an error message with the Nothing constructor. The header is provided.

```
instance Monad Maybe where
```



4. (20 Points) **Parsing.** All elements of lists in Haskell must be of the same type, so it is not possible to have a list where the first item is an `Integer`, the second a `String`, etc. Some other functional languages (e.g. Lisp and Scheme) allow lists with different types, the equivalent of:

```
polyList = ["cat", 3.14159, 'a', ["dog", 5]]
```

However, there is a way around this restriction - define a data type `ListItem` that can contain different types and create a list of `ListItems`. That is, we can say:

```
data ListItem = Str String |
               Intgr Integer |
               Flt Float |
               Chr Char |
               Lst [ListItem]
               deriving Show
```

Then the following would be valid:

```
polyList2 = Lst [Str "cat", Flt 3.14159, Chr 'a', Lst [Str "dog", Intgr 5]]
```

(continued on next page)

You are to create a parser that converts a string in normal list notation into a `ListItem`. Because this is an exam, we will restrict the types of `ListItems` that we parse to `Intgr` and `Lst`. The BNF for the input is:

```
listItem = integer |
          '[' listItem [',' listItem]... ']'
integer = digit [digit]...
```

Assume that you are given a parser `integer :: Parser Integer` which will parse an integer and that there will be no spaces in the input string. Write a parser `listItem :: Parser ListItem` that will parse an input string into list items. 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)
```

Thus a call to

```
listItem "[3,5,[1,2,3],4]"
```

should return

```
Just (Lst [Intgr 3,Intgr 5,Lst [Intgr 1,Intgr 2,Intgr 3],Intgr 4], "")
```

Write the parser `listItem`.



5. (25 Points) **Short Answer.**

- (a) The Sudoku program that we saw in class went through the choice matrix row by row until it found an entry with more than one choice remaining. It then created a new choice matrix for each possible value, thus increasing the number of choice matrices but reducing the number of entries with more than one choice. A suggested exercise was to instead find an entry with the smallest number of choices (greater than 1) and expand that entry, as opposed to taking the first one found.


An argument can be made that this would slow down the program, because it takes time to find the entry with the smallest number of choices and doing this does not reduce the number of solutions considered. The number of solutions considered is the product of the branching factors on all of the levels. Re-ordering which levels have large branching factors and which have small branching factors will not change this product.


What is wrong with this argument?



- (b) We looked at a recursive function for computing the edit distance as a possible way to find “near names” in the Kevin Bacon Game program. The original recursive version was too slow, but an alternate method was much faster. Explain why the first version was slow. Then give the name of and describe the technique that we used to speed up our function for computing edit distance.



(c) Even with the speedup described in the previous part it would be too slow to use this function in a spell checker, where hundreds of thousands of words must be tested. Yet Norvig's spell checker uses edit distance as its primary criterion for selecting nearest words. What clever approach did he use to avoid computing the edit distance all of those times? 

(d) If a type declaration has "deriving Ord" how are two items in that type compared? 

- (e) The queue that we used to implement BFS used two stacks. We had another implementation that used a list. What is the big-O run time to enqueue and dequeue  $n$  items with each implementation? Why?

