

COSC 8, Fall 2007

Final Exam

SAMPLE SOLUTIONS

December 7, 2007

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. ⇐

Solution:

```
factorial = 1 : zipWith (*) factorial [1..]
```

- (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**. ⇐

Solution:

```
evens = 0 : zipWith (+) odds ones
```

```
odds = zipWith (+) evens ones
```

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`



Solution:

```
city recreation name hometown
Hanover biking Tom Hanover
Hanover biking Lisa Hanover
Hanover skiing Tom Hanover
Hanover skiing Lisa Hanover
Miami sunbathing Chris Miami
Miami vice Chris Miami
```

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



Solution:

```
city recreation activity cost
Aspen skiing skiing expensive
Hanover biking biking moderate
Hanover skiing skiing expensive
Las Vegas gambling gambling expensive
Miami sunbathing sunbathing cheap
```

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
```

←←

Solution:

```
(if x >= 0 then return (sqrt x) else fail "sqrt of negative") 4 >>= recipM
= return (sqrt 4) >>= recipM
= Just 2 >>= recipM
= recipM 2
= (if x /= 0 then return (1/x) else fail "division by 0") 2
= return (1/2)
= Just 0.5
```

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

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

←←

Solution:

```
(if x >= 0 then return (sqrt x) else fail "sqrt of negative") (-4) >>= recipM
= fail "Square root of negative number" >>= recipM.
= Nothing >>= recipM
= Nothing
```

- (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
```



Solution:

```
Just x >>= k = k x
Nothing s >>= k = Nothing s
return = Just
fail = Nothing
```

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`.



Solution:

```
listItem = integer >-> Intgr
! lit '[' -# listItem #: takeWhileParse (lit ',' -# listItem) #- lit ']'
>-> Lst
```

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? ⇐

Solution: It ignores pruning. The expectation is that if entries with few choices are expanded first that the entries with many choices will get pruned and reduced to fewer choices. Thus by the time you expand an entry its branching factor will be small. Expanding an entry with a large number of choices means that that level has a large branching factor. The entries with small numbers of choices that are left to be expanded later are less likely to be pruned (fewer choices can cause conflicts), so the overall number of solutions examined is likely to be larger.

- (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. ⇐

Solution: The recursive program was slow because it kept re-solving the same subproblems over and over again. The technique is memoization. The idea is to keep a map of solved problems. When the function is called, the first thing that it does is to see if the problem has been inserted into the map. If so, the answer is looked up in the map and returned immediately. If not, it is computed via recursive calls (which are themselves memoized) and the problem with its answer is inserted into the map. Thus each problem is solved once. After that it is looked up in the map.

- (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? ⇐

Solution: Instead of finding the edit distance between the input string and everything in the dictionary Novig’s program generates all strings that are edit distance 1 and all strings that are edit distance 2 from the input string. It finds the set of strings at edit distance 1 by making all possible edits to the input string. It then maps the same function over all strings of edit distance 1 to get the set of strings of edit distance 2. It intersects these sets with the set of words in the dictionary to determine which are valid words, and thus determines the sets of words at edit distance 1 and 2 without directly computing the edit distance between the input string and any word in the dictionary.

- (d) If a type declaration has “deriving Ord” how are two items in that type compared? ⇐

Solution: They are compared lexicographically. If the constructors are the same, then the arguments to the constructor are compared in the order that they appear. (The arguments must be of type `Ord`, so they can be compared.) If the constructors are different the first one to appear in the `data` declaration is considered to be the smallest.

- (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? ⇐

Solution: The two-stack solution is $O(n)$. Each item is pushed onto each stack once and popped off of each stack once. The list solution is $O(n^2)$. The enqueue operation appends the new item to the end of the list, taking time proportional to the length of the list. If all items are enqueued before any are dequeued this will take $1 + 2 + \dots + n = O(n^2)$ time.