Fall, 1999 Name: ____________________

Com S 541 — Programming Languages 1

Test on Functional Programming Languages

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 5.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write Haskell functions on this test, you may use anything in the Haskell Prelude without writing it in your test. You are encouraged to define functions not specifically asked for if they are useful to your programming; if they are not in the Prelude, write them into your test.

1. (10 points) How does laziness help one write a domain specific embedded language in Haskell? Give at least one example from your own group's language design, or from the article about Fran or Haskore that you read.

2. (5 points) Put a check mark (√) to the left of each one of the following that must be either a constant or a constant function in Haskell. (Recall that a constant function is a function whose output is always the same.) Note: you are supposed to be able to answer this from the information given.

   (a) ________ `random :: Double`
   (b) ________ `changeAssoc :: key -> a -> [(key, a)] -> Maybe [(key, a)]`
   (c) ________ `setGateNumbers :: [(String, Number)] -> ()`
   (d) ________ `todaysDate :: (Int, Int, Int)`
   (e) ________ `updateDB :: (String, Int) -> [Record] -> [Record]`

3. (10 points) Consider the following data type.

```
> data Tree a = Atom a | List [Tree a]
```

Define the function

```
> mapTree :: (a -> b) -> Tree a -> Tree b
```

The following are examples.

```
mapTree (+ 1) (Atom 3) = (Atom 4)
mapTree (+ 1) (List []) = (List [])
mapTree (+ 10) (List [Atom 2, Atom 5, List [Atom 2], List [], Atom 1])
             = (List [Atom 12, Atom 15, List [Atom 12], List [], Atom 11])
mapTree odd (List [Atom 2, Atom 5, List [Atom 2], List [], Atom 1])
      = (List [Atom False, Atom True, List [Atom False], List [], Atom True])
```

Hint: be careful to make sure your solution type checks!

4. (20 points) In Haskell, define the function

```
> smooth :: Double -> [Double] -> [Double]
```

that takes a fraction `delta` such that $0 \leq$ `delta` ≤ 1, and an infinite list of nonnegative numbers, `nums`, and returns a smoothed version of `nums`. Smoothing, means that the i^{th} element of the output is as close to the i^{th} element of `nums` as possible while still satisfying the constraint that it is no more than $1+$ `delta` times the $(i-1)^{\text{th}}$ element of `nums` and no less than $1-$ `delta` times the $(i-1)^{\text{th}}$ element of `nums`. However, the first output equals the first input. For example,

```
smooth 0.5 ([1.0, 3.0, 3.0, 8.0, 8.0] ++ [8, 8 ..])
          = [1.0, 1.5, 3.0, 4.5, 8.0] ++ [8, 8 ..]
smooth 0.05 ([1, 2.00, 3.0, 4.00, 4.0] ++ [4, 4 ..])
           = [1 ,1.05, 2.1, 3.15, 4.0] ++ [4, 4 ..]
smooth 0.05 ([4, 4, 3.0, 2.00, 1.0, 5.00, 5.0] ++ [5, 5 ..])
           = [4, 4, 3.8, 2.85, 1.9, 1.05, 5.0] ++ [5, 5 ..]
smooth 0.5 ([4, 4, 3.0, 2.00, 1.0, 5.0, 5.0] ++ [5, 5 ..])
          = [4, 4, 3.0, 2.00, 1.0, 1.5, 5.0] ++ [5, 5 ..]
smooth 0.9 ([4.0, 20.0, 1.0, 8.0, 32.0, 4.0, 5.0] ++ [5, 5 ..])
          = [4.0, 7.60, 2.0, 1.9, 15.2, 4.0, 5.0] ++ [5, 5 ..]
```

5. (20 points) This is a problem about type checking. It is adapted from Perry Alexander's specification language for VHDL, which uses a small constraint language to state constraints on time, space, and power consumption.

   Consider the following data definitions.

```
> data Constraint = Compare Resource BOp Measure
>                 | Constraint `And` Constraint
> data Resource = TimeUsed | SpaceUsed | PowerUsed
>                     deriving Eq
> data Measure = Microsecond | Word | MilliWatt | Int :*: Measure
>                     deriving Eq
> data BOp = Less | LessEq | Greater | GreaterEq | Equal
```

   In such a language, is important not to confuse different units of measure; in particular when comparing a resource and measure, both should have the same units. For example, each of the following lines should be a type error:

```
Compare TimeUsed Less (6 :*: Word)
Compare SpaceUsed Less (7 :*: MilliWatt) `And` Compare PowerUsed Less Word
```

   Write code for the Haskell function

```
> typecheck :: Constraint -> Maybe ()
```

   where the result `Just ()` means that the constraint has successfully type checked, and the result `Nothing` means that a type error was found. The following are samples.

```
typecheck (Compare TimeUsed Less (6 :*: Word)) = Nothing
typecheck (Compare TimeUsed Less (6 :*: Microsecond)) = Just ()
typecheck (Compare SpaceUsed Less (5 :*: (4 :*: Word))
          `And` Compare PowerUsed Greater MilliWatt) = Just ()
```

6. (25 points) A wealthy client (okay, it's the US Navy), wants you to head a team that will write many programs to keep track of potentially infinite geometric regions.

   Your task is to design a domain specific language embedded in Haskell for this, assuming that the type of geometric regions is specified as follows. (The type `Point` is abstract; you don't need to know anything about it for this problem.)

```
> type Region = Point -> Bool
```

   Design and implement a small set of primitives that can be used to construct and manipulate `Region` values from within Haskell programs. For each primitive, give the type, the code, and if necessary, some comments telling what it is supposed to do. (The comment can get you partial credit if you cannot write the code, words the code is wrong.) You should have at least five other primitives.

7. (10 points) Briefly answer the following question. Are there any primitives on Regions that are made difficult by the type `Region` given above? If so, give an example; if not, say why not.