Homework 1: Functional Programming, Haskell

Due: problems 1 and 3, Thursday, September 1, 2005; problems 4–9, Thursday, September 15, 2005; remaining problems Thursday, September 29, 2005.

Note that you may want to be thinking about problems 23 and 24 during the entire homework.

In this homework you will learn: the basics of Haskell, how Haskell can be considered as a domain-specific language for working with lists, basic techniques of recursive programming over various types of data, and abstracting from patterns, higher-order functions, currying, and infinite data. Many of the problems below exhibit polymorphism. The problems as a whole illustrate how functional languages work without hidden side-effects.

Since the purpose of this homework is to ensure skills in functional programming, this is an individual homework. That is, for this homework, you are to do the work on your own (not in groups).

For all Haskell programs, you must run your code with Haskell 98 (for example, using hugs). We suggest using the flags +t -u when running hugs. A script that does this automatically is provided in the course bin directory, i.e., in /home/course/cs541/public/bin/hugs, which is also available from the course web site. If you get your own copy of Haskell (from http://www.haskell.org), you can adapt this script for your own use.

You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and the output of your testing, for all questions that require code.

Be sure to clearly label what problem each function solves with a comment.

Read Thompson’s book, Haskell: The Craft of Functional Programming (second edition), chapters 1–7, 9–14 and 16–18. You may also want to read a tutorial on the concepts of functional programming languages, such as Hudak’s computing survey article mentioned in the “Introduction to the Literature” handout, or the “Gentle Introduction to Haskell” (which is on-line at haskell.org) for a different introduction to the language.

Some of the problems build on each other. Don’t hesitate to contact the staff if you are stuck at some point.

Acknowledgment: many of these problems are due to John Hughes.

1. (30 points) Write a function

> delete_all :: (Eq a) => a -> ([a] -> [a])

that takes an item (of a type that is an instance of the Eq class) and a list, and returns a list just like the argument list, but with the each occurrence of the item (if any) removed. For example.

delete_all 3 [] = []
delete_all 1 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [2, 3, 2, 2, 3, 2]
delete_all 4 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 1, 2, 3, 2, 1]
delete_all 3 [1, 2, 3] = [1, 2]

Do this (a) using a list comprehension, and (b) by just using functions in the Haskell Prelude, which is in the file /opt/hugs/lib/hugs/libraries/Prelude.hs, assuming hugs is installed in /opt/hugs. (c) by writing out the recursion yourself.

2. (suggested practice) Write a function

> delete_second :: (Eq a) => a -> ([a] -> [a])

1
that takes an item (of a type that has an == function defined for it) and a list, and returns a list just like the argument list, but with the second occurrence of the item (if any) removed. For example.

```
delete_second 3 ([]:[Int]) = [] :: [Int]
delete_second 1 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 2, 3, 2, 1]
delete_second 4 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 1, 2, 3, 2, 1]
delete_second 3 [1, 2, 3] = [1, 2, 3]
```

Do this both (a) by just using functions in the Haskell Prelude, and (b) by writing out the recursion yourself. (Can this be done using a list comprehension?)

Hint: for part (b), you may need a helping function.

3. (10 points) In Haskell, write a function

```
> associated :: (Eq a) => a -> [(a,b)] -> [b]
```

such that associated x pairs is the list, in order, of the second elements of pairs in pairs, whose first element is equal to the argument x.

For example:

```
associated 3 ([]:[(Integer,Float)]) = []
associated 3 [(3,4), (5,7), (3,6), (9,3)] = [4, 6]
associated 2 [(1,'a'), (3,'c'), (2,'b'), (4,'d')] = ['b']
associated 'c' (zip ['c', 'c' ..] [1, 2 ..]) = [1, 2 ..]
```

Do this (a) using a list comprehension, and (b) using functions in the Haskell prelude.


5. The following relate to modularization of numeric code using functional techniques and lazy evaluation (you should read chapter 17 in Thompson's book about laziness). In particular, we will explore the Newton-Raphson algorithm. This algorithm computes better and better approximations to the square root of a number \( n \) from a previous approximation \( x \) by using the following function.

```
> next :: (Real a, Fractional a) => a -> a -> a
> next n x = (x + n / x) / 2
```

(a) (10 points) Using the iterate function in the Haskell Prelude, write a function

```
> approximations :: (Real a, Fractional a) => a -> a -> [a]
```

such that approximations \( n \ a0 \) returns the infinite list of approximations to the square root of \( n \), starting with \( a0 \). For example,

```
approximations 1.0 1.0 = [1.0, 1.0 ..]
take 5 (approximations 2.0 1.0) = [1.0, 1.5, 1.41667, 1.41422, 1.41421]
take 5 (approximations 64.0 1.0) = [1.0, 32.5, 17.2346, 10.474, 8.29219]
```

(b) (20 points) Define a function within
> \textit{within} :: (\textit{Ord} \ a, \textit{Num} \ a) \Rightarrow \ a \rightarrow \ [\ a] \rightarrow \ a

that takes a tolerance, that is, a number \textit{epsilon}, and an infinite list of numbers, and looks down the list to find two consecutive numbers in the list that differ by no more than \textit{epsilon}; it returns the second of these. (It might never return if there is no such pair of consecutive elements.) For example,

\begin{align*}
\text{within} \ 1.0 \ [1.0 \ ..\ ] &= 2.0 \\
\text{within} \ 0.5 \ ([1.0, \ 32.5, \ 17.2346, \ 10.474, \ 8.29219, \ 8.00515] \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ qua
that takes an initial value for \(\delta\), and the function \(f\), and a point \(x\), and returns an infinite list of approximations to the derivative of \(f\) at \(x\), where at each step, the current \(\delta\) is halved. For example

\[
\text{take 9 (diffApproxims 500.0 (\(x \rightarrow x^2\)) 20)} = [540.0, 290.0, 165.0, 102.5, 71.25, 55.625, 47.8125, 43.9062, 41.9531]
\]

\[
\text{take 8 (diffApproxims 100.0 (\(x \rightarrow x^3\)) 10)} = [13300.0, 4300.0, 1675.0, 831.25, 526.562, 403.516, 349.316, 324.048]
\]

7. (15 points) Write a function

\[
> \text{differentiate} :: (\text{Real a}, \text{Fractional a}) \Rightarrow a \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]

that takes a tolerance, \(\varepsilon\), an initial value for \(\delta\), and the function \(f\), and a point \(x\), and returns an approximation to the derivative of \(f\) at \(x\). For example.

\[
\text{differentiate 0.1e-6 500.0 (\(x \rightarrow x^2\)) 20 = 40.0}
\]

\[
\text{differentiate 0.1e-6 100.0 (\(x \rightarrow x^3\)) 10 = 300}
\]

(Hint: use the answer to the previous problem.)

8. (30 points; extra credit) Write a function in Haskell to do numerical integration, using the ideas above.

9. (15 points) Write a function

\[
> \text{compose} :: [(a \rightarrow a)] \rightarrow (a \rightarrow a)
\]

that takes a list of functions, and returns a function which is their composition. For example.

\[
\text{compose [] [1, 2, 3] = [1, 2, 3]}
\]

\[
\text{compose [[\(x \rightarrow x + 1\)], (\(x \rightarrow x + 2\))] 4 = 7}
\]

\[
\text{compose [tail, tail, tail] [1, 2, 3, 4, 5] = [4, 5]}
\]

\[
\text{compose [[\(x \rightarrow 3 : x\)], (\(y \rightarrow 4 : y\))] [] = 3 : (4 : [])}
\]

Hint: note that \(\text{compose} []\) is the identity function.

10. (10 points) Write a function

\[
> \text{merge} :: (\text{Ord a}) \Rightarrow \text{[[a]]} \rightarrow \text{[a]}
\]

that takes a finite list of sorted finite lists and merges them into a single sorted list. A “sorted list” means a list sorted in increasing order (using \(<\)); you may assume that the sorted lists are finite. For example

\[
\text{merge [[[]]]::[[Int]]] = [] :: [Int]
\]

\[
\text{merge [[1, 2, 3]] = [1, 2, 3]}
\]

\[
\text{merge [[1, 3, 5, 7], [2, 4, 6]] = [1, 2, 3, 4, 5, 6, 7]}
\]

\[
\text{merge [[1, 3, 5, 7], [2, 4, 6], [3, 5, 9, 10, 11, 12]] = [1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12]}
\]

\[
\text{take 8 (merge [[1, 3, 5, 7], [1, 2, 3, 4, 5, 6, 7, 8]]) = [1, 1, 2, 3, 3, 4, 5, 5]}
\]
11. (10 points, extra credit) Consider the following type as a representation of binary relations.

\[ \text{type BinaryRel a b = [(a, b)]} \]

(a) (10 points, extra credit) Write a function

\[ \text{isFunction :: (Eq a, Eq b) => (BinaryRel a b) -> Bool} \]

that returns True just when its argument satisfies the standard definition of a function; that is, \( \text{isFunction } r \) is True just when for each pair \((x, y)\) in the list \( r \), there is no pair \((x, z)\) in \( r \) such that \( y \neq z \).

(b) (10 points, extra credit) Write a function

\[ \text{brelCompose :: (Eq a, Eq b, Eq c) => (BinaryRel a b) -> (BinaryRel b c) -> (BinaryRel a c)} \]

that returns the relational composition of its arguments. That is, a pair \((x, y)\) is in the result if and only if there is a pair \((x, z)\) in the first relation argument of the pair of arguments, and a pair \((z, y)\) in the second argument. For example,

\[
\begin{align*}
\text{brelCompose } ([()] :: [(Int, Int)]) &\cdot [(2, 'b'), (3, 'c')] = [] \\
\text{brelCompose } ([()] :: [(Int, Int)]) &\cdot [(1, 'b'), (2, 'c')] = [] \\
\text{brelCompose } [(1, 2), (2, 3)] &\cdot [(2, 'b'), (3, 'c')] = [(1, 'b'), (2, 'c')] \\
\text{brelCompose } [(1, 2), (1, 3)] &\cdot [(2, 'b'), (3, 'c')] = [(1, 'b'), (1, 'c')] \\
\text{brelCompose } [(1, 3), (2, 3)] &\cdot [(3, 'b'), (3, 'c')] \\
&= [(1, 'b'), (1, 'c'), (2, 'b'), (2, 'c')] \\
\end{align*}
\]

12. (5 points) Define a function

\[ \text{commaSeparate :: [String] -> String} \]

that takes a list of strings and returns a single string that contains the given strings in the order given, separated by ", "

For example,

\[
\begin{align*}
\text{commaSeparate } [] &= "" \\
\text{commaSeparate } ["a", "b"] &= "a, b" \\
\text{commaSeparate } ["Monday", "Tuesday", "Wednesday", "Thursday"] \\
&= "Monday, Tuesday, Wednesday, Thursday"
\end{align*}
\]

13. (10 points) Define a function

\[ \text{onSeparateLines :: [String] -> String} \]

that takes a list of strings and returns a single string that, when printed, shows the strings on separate lines. Do this both (a) using functions in the prelude, and (b) defining it explicitly using recursion.

Hint: if you want your tests to show items on separate lines, use Haskell’s \text{putStr} function in your testing. For example,
Main> putStr (onSeparateLines ["mon","tues","wed"])
mon
tues
wed :: IO()

14. (10 points) Define a function

> separatedBy :: String -> [String] -> String

That is a generalization of onSeparateLines and commaSeparated. Test it by using it to define these other functions.

15. (5 points) Redefine the ++ operator on lists using foldr, by completing the following module by adding arguments to foldr. You’ll find the code for foldr in the Haskell prelude. (Hint: you can pass the “.” constructor as a function by writing (:).)

    module MyAppend where
    import Prelude hiding ((++))
    (++) :: [a] -> [a] -> [a]
    xs ++ ys = foldr _________________________

16. (10 points) Define the function

> doubleAll :: [Integer] -> [Integer]

that takes a list of Integers, and returns a list with each of its elements doubled. Do this (a) using a list comprehension, and (b) using foldr in a way similar to the previous problem. (Hint: use a where to define the function you need to pass to foldr. You might want to use function composition, written with an infix dot (.) in Haskell.) The following are examples.

doubleAll [] = []
doubleAll [8,10] = [16,20]
doubleAll [1, 2 .. 500] = [2, 4 .. 1000]

17. (15 points) (a) Define the map functional using foldr. As part of your testing, use map to (a) define doubleAll, and (b) to add 1 to all the elements of a list of Integers. (Hint: import the Prelude hiding map in the module for this answer; see problem 15 above.)

18. Consider the following data type for trees, which represents a Tree of type a as a Node, which contains an item of type a and a list of Trees of type a.

> data Tree a = Node a [Tree a] deriving Show

(a) (10 points) Define a function sumTree

> sumTree :: Tree Integer -> Integer

which adds together all the Integers in a Tree of Integers. For example,

sumTree (Node 4 []) = 4
sumTree (Node 3 [Node 4 [], Node 7 []]) = 14
sumTree (Node 10 [Node 3 [Node 4 [], Node 7 []],
                   Node 10 [Node 20 [], Node 30 [], Node 40 []]]) = 124
(b) (10 points) Define a function `preorderTree`:

```haskell
> preorderTree :: Tree a -> [a]
```

which takes a Tree and returns a list of the elements in its node in a preorder traversal. For example,

- `preorderTree (Node True []) = [True]`
- `preorderTree (Node 5 [Node 6 [], Node 7 []]) = [5, 6, 7]`
- `preorderTree (Node 10 [Node 3 [Node 4 [], Node 7 []],
  Node 10 [Node 20 [], Node 30 [], Node 40 []]]) = [10, 3, 4, 7, 10, 20, 30, 40]`

(c) (15 points) Give a Haskell `instance` declaration (see chapter 12) that makes `Tree` an instance of the type class `Functor`. (See the Prelude for the definition of the `Functor` class.) Your code should start as follows:

```haskell
instance Functor Tree where
  fmap f ... 
```

(d) (30 points) By generalizing your answers to the above problems, define a Haskell function `foldTree`:

```haskell
> foldTree :: (a -> b -> c) -> (c -> b -> b) -> b -> Tree a -> c
```

that is analogous to `foldr` for lists. This should take a function to replace the `Node` constructor, one to replace the `(:)` constructor for lists, and a value to replace the empty list. You should, for example, be able to define `sumTree`, `preorderTree`, and the Functor instance for `fmap` on Trees as follows:

```haskell
> sumTree = foldTree (+) (+) 0
> preorderTree = foldTree (:) (++) []
> instance Functor Tree where
>   fmap f = foldTree (Node . f) (:) []
```

19. (30 points) A set can be described by a “characteristic function” (whose range is the booleans) that determines if an element occurs in the set. For example, the function \(\phi\) such that \(\phi(\text{"coke"}) = \phi(\text{"pepsi"}) = \text{True}\) and for all other arguments \(x\), \(\phi(x) = \text{False}\) is the characteristic function for a set containing the strings "coke", "pepsi" and nothing else. Allowing the user to construct a set from a characteristic function gives one the power to construct sets that may “contain” an infinite number of elements (such as the set of all prime numbers).

Let the polymorphic type constructor `Set` be some polymorphic type that you decide on (you can declare this with something like the following):

```haskell
type Set a = ...
-- or perhaps something like --
data Set a = ...
```

Hint: think about using a function type.

The operations on sets are described informally as follows.

(a) The function
setSuchThat :: (a -> Bool) -> (Set a)

takes a characteristic function, \( f \) and returns a set such that each value \( x \) (of appropriate type) is in the set just when \( fx \) is \textbf{True}.

(b) The function

\textbf{unionSet} :: Set a -> Set a -> Set a

takes two sets, with characteristic functions \( f \) and \( g \), and returns a set such that each value \( x \) (of appropriate type) is in the set just when \( (fx) \) or \( (gx) \) is true.

(c) The function

\textbf{intersectSet} :: Set a -> Set a -> Set a

takes two sets, with characteristic functions \( f \) and \( g \), and returns a set such that each value \( x \) (of appropriate type) is in the set just when both \( (fx) \) and \( (gx) \) are true.

(d) The function

\textbf{memberSet} :: Set a -> a -> Bool

tells whether the second argument is a member of the first argument.

(e) The function

\textbf{complementSet} :: Set a -> Set a

which returns a set that contains everything (of the appropriate type) not in the original set.

As examples, consider the following.

\begin{verbatim}
memberSet (setSuchThat (\ x -> x == "coke")) "coke" = True
memberSet (setSuchThat (\ x -> x == "coke")) "pepsi" = False
memberSet (complementSet (setSuchThat (\ x -> x == "coke"))) "coke" = False
memberSet (setSuchThat (\ x -> x == "coke"))
(memberSet (\ x -> x == "pepsi"))
"pepsi" = True
memberSet (unionSet (setSuchThat (\ x -> x == "coke"))
(memberSet (\ x -> x == "pepsi")))
"coke" = True
memberSet (unionSet (setSuchThat (\ x -> x == "coke"))
(memberSet (\ x -> x == "pepsi")))
"sprite" = False
memberSet (intersectSet (setSuchThat (\ x -> x == "coke"))
(memberSet (\ x -> x == "pepsi")))
"coke" = False
\end{verbatim}

Note (hint, hint) that the following equations must hold, for all \( f \), \( g \), and \( x \) of appropriate types.

\begin{verbatim}
memberSet (unionSet (setSuchThat f) (setSuchThat g)) x = (f x) || (g x)
memberSet (intersectSet (setSuchThat f) (setSuchThat g)) x = (f x) && (g x)
memberSet (setSuchThat f) x = f x
memberSet (complementSet (setSuchThat f)) x = not (f x)
\end{verbatim}

20. (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.
> type Point = (Int, Int)
> 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. You should have at least five other primitives. Make at least one of them work on 2 or more regions in some way.

21. (25 points) Consider the following data definitions.

> data Exp = BoolLit Bool | IntLit Integer | CharLit Char
>          | Sub Exp Exp
>          | Equal Exp Exp
>          | If Exp Exp Exp
> data OType = OBoolean | OInt | OChar | OWrong
>            deriving (Eq, Show)

Write a function

> typeOf :: Exp -> OType

that takes an Exp and returns its OType. For example.

typeOf (Equal (IntLit 3) (IntLit 4)) = OBoolean
typeOf (Sub (IntLit 3) (IntLit 4)) = OInt
typeOf (Sub (CharLit 'a') (IntLit 4)) = OWrong
typeOf (If (BoolLit True) (IntLit 4) (IntLit 5)) = OInt

Your program should incorporate a reasonable notion of what the exact type rules are. (Exactly what “reasonable” is left up to you; explain any decisions you feel the need to make.)

22. (10 points) Based on the types below, which of each one of the following 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, regardless of its arguments.) 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]

23. Pick a language that you know well (e.g., Java, C++, C#, Fortran, or C). In this problem you will compare Haskell and that language.

For each of the following, answer the question clearly, using at least one concrete programming example (i.e., write code, at least in Haskell, but preferably in both languages, to illustrate the comparison).
(a) (20 points) What kinds of programming tasks does Haskell automate better than that language?

(b) (20 points) In what ways does Haskell permit better abstraction than that language?

(c) (20 points) In what ways does Haskell support code reuse better than that language?

24. (30 points) Based on your answer to the previous problem, list (all) the features of Haskell’s design that you would like to include in the design of “the next programming language.” For each feature, briefly state why it would be helpful to include that in a language that would be a successor to Java.

25. (30 points; extra credit) What features of Haskell’s design would you omit from the design of “the next programming language?” For each feature, briefly explain your reasons.

26. (20 points; extra credit) What features of Haskell do you think would be difficult for the average programmer to understand and use on a daily basis (even with reasonable tutorial materials and training)?

27. (50 points total; extra credit) Read either one of the following two articles:


or some other published research article in a journal or conference proceedings on implementing domain-specific languages embedded in Haskell. (By a published research article, I mean an article that is not in a trade journal (e.g., it has references at the end), and that is from a refereed journal or conference. Publication means the article actually appeared in print, and was not just submitted somewhere. So beware of technical reports on the web. It’s okay to get a copy of a published article from the web, although I highly encourage you to physically go to the library.)

Write a short (1 or 2 page maximum) review of the article, stating:

- (10 points) what the problem was that the article was claiming to solve,
- (20 points) the main points made in the article and what you learned from it,
- (20 points) what contribution it make vs. any related work mentioned in the article.

In your writing, be sure to digest the material; that is, don’t just select various quotes from the article and string them together, instead, really summarize it. If you quote any text from the paper, be sure to mark the quotations with quotation marks (“ and ”) and give the page number(s).

If you do a different article than one of the two mentioned above, then hand in a copy of the article with your review.

28. (10 points; extra credit) How does laziness help one write a domain specific embedded language in Haskell? Give at least one example.