Mid-term examination #1 — given Tuesday 29th September

General instructions

Online, take-home exam.

Deadline to submit via email: 9:00 pm.

Total points available: 150 pts.

Answer in the spaces provided.

Your name (print):

I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.

Please sign and date:

1.1 Haskell—types and evaluation (40 pts)

For each of the following Haskell expressions (5 pts each):

- State whether the expression is well-typed or not. **If it is not well-typed, provide an explanation for your answer.**
- If the expression is well-typed, provide its most general type and then indicate whether evaluation of the expression reduces to a value, loops forever, or raises a runtime error. **Provide an explanation for your answer.**
- If the expression is well-typed and reduces to a value, write that value. If the value of the expression is a function, explain that function's behavior.

Assume that integer constants such as `3`, `4`, `5` all have type `Int`.

1. `\m -> \n -> head (n ++ m)`

2. `(\m -> \n -> head (n ++ m)) [3, 9] [4, 5]`

3. `(\m -> \n -> head (n ++ m)) []`

4. `(\m -> \n -> head (n ++ m)) [] []`

5. `(\m -> \n -> head (n ++ m)) 3 (4, 5)`

6. `f 8`

where `f` is defined as follows:

```
f x | x `mod` 2 == 0 = 1 + f (x+1)
```

7. `g 8`

where `g` is defined as follows:

```
g x | x `mod` 2 == 0 = 1 + g (x+1)
    | x `mod` 2 == 1 = 1 + g (x-1)
```

8. `foldr (+) 12 (takeWhile (<10) [2,4..])`

1.2 Haskell—list functionals (30 pts)

1. (10 pts) Name and briefly explain the feature of Haskell evaluation means that Haskell programs can manipulate infinite lists without necessarily looping forever.

   Illustrate your answer with an example.

2. (10 pts) Under what conditions might it be appropriate to use a return type of the form `Either a b` in a Haskell function?

   Illustrate your answer with an example.

3. (10 pts) Briefly explain how Haskell handles pattern matching in function definitions with multiple clauses.

   Illustrate your answer by considering the behavior of the following functions, `f` and `g`:

```
f [] = 0
f xs = 1 + f (drop 1 xs)

g xs = 1 + g (drop 1 xs)
g [] = 0
```

   when passed both empty and non-empty lists as arguments.

1.3 Haskell—tree datatypes (40 pts)

Here is the definition of a Haskell type of binary trees with exactly two sub-trees per node (with useful data values stored in the leaves but not in the interior nodes):

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

where the first and second arguments to `Node` represent the "left" and "right" sub-trees of that node, respectively.

1. (10 pts) Write a Haskell function

   ```
   mirror :: Tree a -> Tree a
   ```

   that mirrors the tree about the root node. For instance,

   ```
   mirror (Node (Leaf 1) (Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)))
   ```

   should evaluate to

   ```
   Node (Node (Leaf 4) (Node (Leaf 3) (Leaf 2))) (Leaf 1)
   ```

2. (15 pts) Write a Haskell function

```
treeFilter :: (a -> Bool) -> Tree a -> Tree (Maybe a)
```

that takes as its arguments a predicate `p` and a binary tree `t` and returns a new tree with the same shape as `t` in which every leaf value `v` from `t` for which `p v` is `False` is replaced by `Nothing` and every leaf value `v` from `t` for which `p v` is `True` is replaced by `Just v`. For example,

```
treeFilter (\n -> n `mod` 2 == 0)
           (Node (Leaf 1) (Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)))
```

should evaluate to

```
Node (Leaf Nothing) (Node (Node (Leaf (Just 2)) (Leaf Nothing)) (Leaf (Just 4)))
```

3. (15 pts) Write a Haskell function

```
treeMax :: (Ord a) => Tree a -> a
```

that returns the largest data value stored in the tree. In this type signature, the type constraint "`(Ord a) =>`" just means that you may assume the existence of an ordering using `>`, `>=`, `max` etc on the parameter type `a`. For example,

```
treeMax (Node (Leaf 1) (Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)))
```

should evaluate to `4`.

1.4 Proving properties of programs (40 pts)

Consider a function `reverse :: [a] -> [a]` that reverses a list, i.e.:

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Now prove, by induction on lists, that

```
foldr (\x -> \acc -> acc ++ [f x]) [] l  =  reverse (map f l)
```

for all `l :: [a]` and for all `f :: a -> b`.