CSE 114A Midterm 2, Winter 2024

NAME : _____

CruzID: _____@ucsc.edu

- DO NOT TURN THIS PAGE OVER BEFORE WE TELL YOU TO
- You have **90 minutes** to complete this exam.
- Where limits are given, **write no more** than the amount specified.
- You may refer to a **double-sided cheat sheet**, but no electronic materials.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- If you are unsure of how to interpret a problem description, state your interpretation clearly and concisely. Reasonable interpretations will be taken into acount by the graders.
- **Good luck!**

Q1: Scope

What are the free variables of this λ-term?

```
(\b -> a (\a -> c) (\d e -> a d)) (\h i -> g)
```

Answer: _____

Q2: Reductions

Evaluate this λ-term to a normal form. Reminder:

- `=a>` stands for an α-step (α-renaming)
- `=b>` stands for a β-step (β-reduction)

```
(\z x -> x z) (x y)
```

=a> _____

=b> _____

Q3: Haskell filter

What does this Haskell expression evaluate to? (See Haskell cheat sheet for the definition of filter.)

```
filter  (\(x,y) -> x < y)   [(0,5), (4,3), (4,5)]
```

Answer: _____

Q4: Haskell map

What does this Haskell expression evaluate to? (See Haskell cheat sheet for the definition of map.)

```
map (\(x,y) -> x + y)   [(0,1), (2,3), (4,5)]
```

Answer: _____

Q5: Haskell fold 1

What does this Haskell expression evaluate to? (See Haskell cheat sheet for the definition of foldr.)

```
foldr (:) [(0,0)]   [(0,1), (2,3), (4,5)]
```

Answer: _____

Q6: Haskell fold 2

What does this Haskell expression evaluate to? (See Haskell cheat sheet for the definition of foldr.)

```
foldl (-) 10   [1,2,3]
```

Answer: _____

Q7: Haskell fold 3

What does this Haskell expression evaluate to? (See Haskell cheat sheet for the definition of foldr.)

```
foldr (-) 10   [1,2,3]
```

Answer: _____

Q8: Haskell data types

Consider the datatype below for a tree.

```
data Tree = Leaf
          | Node Int Tree Tree
```

Complete the following function definition so that it returns the maximum integer in a tree, or returns 0 for the empty tree. (The function max returns the maximum of its two integer arguments.)

```
maxIntTree :: Tree -> Int


maxIntTree Leaf         =


maxIntTree (Node n l r) =
```

Q9: Haskell data types (continued)

Complete the following function definition so that it converts a Tree into a String matching the following examples. Pay attention to including the right parentheses and spacing. Remember that (++) concatenates Strings, and (show n) converts an Int n into a String. It is ok if your code splits up over multiple lines.

```
-- Examples:
-- treeToList Leaf returns "Leaf"
-- treeToList (Node 4 Leaf Leaf)  returns "(Node 4 Leaf Leaf)"
-- treeToList (Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf))
--   returns "(Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf))"

treeToList :: Tree -> String


treeToList Leaf         =


treeToList (Node n l r) =
```

Q10: Haskell data types (continued)

Consider the function:

```
m :: (Int -> Int) -> Tree -> Tree
m f Leaf = Leaf
m f (Node n l r) = Node (f n) (m f l) (m f r)
```

What does the following expression evaluate to?

```
m (\x -> x*2+1) (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
```

Answer: _____

Q11: Interpreters

Consider the following (strange!) interpreter for a small language:

```
data Exp = Num Int
         | Add Exp Exp
         | Mul Exp Exp

eval :: Exp -> Int
eval (Num n) = n
eval (Add e1 e2) = (eval e1) * (eval e2)
eval (Mul e1 e2) = (eval e1) - (eval e2)
```

What does the following expression evaluate to?

```
eval (Add (Mul (Num 10) (Num 5)) (Num 10))
```

Answer: _____

Haskell Cheat Sheet

Here is a list of definitions you may find useful:

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b xs          = helper b xs
  where
    helper acc []     = acc
    helper acc (x:xs) = helper (f acc x) xs

filter :: (a -> Bool) -> [a] -> [a]
filter pred []    = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys

even :: (Integral a) => a -> Bool
(==) :: Eq a => a -> a -> Bool
max  :: Ord a => a -> a -> a
(<)  :: Ord a => a -> a -> Bool
(>)  :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
```