1) assign points.

CSE 114A Final, Winter 2024

NAME : Priyanka

CruzID: ______________________________@ucsc.edu

- DO NOT TURN THIS PAGE OVER BEFORE WE TELL YOU TO
- You have **180 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.
- The back side of each page will not be scanned or graded, but you can use it as scratch paper if you like.
- **Good luck!**

Q1: Scope

What are the free variables of this λ-term?

`(\a d -> b (\b -> c a)) d`

- [ ] No free variables
- [ ] `a, b, c, d`
- [x] `b, c, d`
- [ ] `a, c, d`
- [ ] `a, b, d`
- [ ] `a, b, c`
- [ ] `b, c`

Q2: Normal Forms

Which of the following terms are in normal form:

- [x] `x`
- [x] `x y`
- [ ] `(\x -> x) y`
- [x] `x (\y -> y)`
- [ ] `(\y -> y y)  (\y -> y y)`

Q3: Reductions

Evaluate this λ-term to a normal form, and for each reduction write in the appropriate blank space whether it is an `=a>` step (α-renaming) or a `=b>` step (β-reduction).

```
(\f x -> f (f x)) (\y -> x y) three
```

=a> (λf z → f (f z)) (λy → x y) three

=b> (λz → (λy → x y) ((λy → x y) z)) three

=b> (λy → x y) ((λy → x y) three)

=b> (λy → x y) (x three)

=b> x (x three)

=___> ____________________

Q4: Haskell map and foldl

What does this Haskell expression evaluate to?

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

Answer: 15

0 2
2 3
4 5
6 10

Q5: Haskell map and foldr

What does this Haskell expression evaluate to?

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

Answer: 12

Q6: Haskell foldr

What does this Haskell expression evaluate to?

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

Answer: (6, 10)

Q7: Haskell foldl 1

In the context of the following function definition:

```
foo f = foldl (\b x -> (f x) : b) []
```

what does this Haskell expression evaluate to?

```
foo (\x -> x+1) [1,2,3]
```

Answer: [4, 3, 2]

Q8: Haskell foldl 2

In the context of the same function definition above, what does this Haskell expression evaluate to?

```
foo (\x -> x+1) (foo (\x -> x+1) [1,2,3])
```

Answer: [3, 4, 5]

Q9: Haskell filter

What does this Haskell expression evaluate to?

```
filter   (\(x,y) -> x y)   [(even,1),  (\x->x>2,3),  (even,6)]
```

Answer: [(λx → x > 2, 3), (even, 6)]

foo

9

Q10: Haskell data types

Consider the following datatype definition:

```
data Paragraph = Text String | Heading Int String | Bullets Bool [String]
```

What is the type of `Heading`?

`Heading ::` Int → String → Paragraph.

Q11: Haskell data types (continued)

In the context of the above datatype defintion, what is the type of

```
(\p ->
  case p of
    Text x -> x
  | Heading x y -> y)
```

Answer: Paragraph → String.

Q12: Haskell data types (continued)

Consider the datatype below for a tree.

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

Write a function that sums all the integer in a trees.

Your solution should not be recursive, and instead should be defined in terms of the `foldTree` function below, by passing in the correct three arguments to `foldTree`.

```
foldTree :: (Int -> Int -> Int -> Int) -> Int -> Tree -> Int
foldTree op base Leaf = base
foldTree op base (Tree n l r) = op n (foldTree op base l) (foldTree op base r)
```

```
sumTree :: Tree -> Int
```

sumTree t = foldTree (\n l r -> n + l + r) 0 t

Q13: Haskell data types (continued)

The function `foldTree` above has a more general/polymorphic type. What is it?

Answer: `foldTree ::` (a -> b -> b -> b) -> b -> Tree -> b

or Int as Tree is defined with Int.

Q14: Haskell data types (continued)

Write an instance of the type class `Show` for the datatype `Tree` according to the following examples. Pay attention to including the correct 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:
-- show Leaf returns "Leaf"
-- show (Node 4 Leaf Leaf)  returns "(Node 4 Leaf Leaf)"
-- show (Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf))
--   returns "(Node 3 (Node 4 Leaf Leaf) (Node 5 Leaf Leaf))"

instance Show Tree where
    show Leaf = "Leaf"
    show (Node n l r) = "(Node " ++ show n
        ++ show l ++ " " ++ show r
        ++ ")"
```

Q15: Implementing An Interpreter

Fill in the blanks in the following Haskell code to define an evaluator for this small programming language:

```
data Expr = Num Int
          | Add Expr Expr

eval :: Expr -> Int

eval (Num n)     = ___n______________

eval (Add e1 e2) = ___(eval e1) + (eval e2)___
```

Q16: Free Variables

For the following small language, define a function `free` so that `free x e` determines if the variable x occurs free in the expression e.

```
type Id = String
data Expr = Num Int
          | Lam Id Expr
          | Var Id
          | App Expr Expr

free:: Id -> Expr -> Bool
```

```
free x (Num n) = False
free x (Lam id e) = (x /= id) && (free x e)
free x (Var id) = (x == id)
free x (App e1 e2) = (free x e1) || (free x e2)
```

Q17: Static and Dynamic Scope

Consider the following code fragment:

```
let euroToUSD = 1.1 in
let convertEurosToDollars eu = eu * euroToUSD in
let euroToUSD = 0.9 in
convertEurosToDollars 100
```

What would this code evaluate to under:

- Static (aka Lexical) Scope: 110
- Dynamic Scope: 90

Q18: Type Systems

What is the result of applying the substitition `U = [a / Int, c / Int]` to the type `(Int -> a -> b)`

Answer: Int -> Int -> b

Q19: Type Systems (continued)

What substitution is the most general unifier of the following two types:

- `(Int -> b)`
- `(a -> Int -> Int)`

Answer: [a / Int, b / Int -> Int]

Haskell Cheat Sheet

Here is a list of definitions you may find useful:

```
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
```