CSE 116, Fall 2019 Midterm

Section	**Points**	**Score**
Part I	40 points	
Part II	56 points	
Total	96 points	

Instructions

- **You have 95 minutes to complete this exam.**
- This exam is **closed book**. You may use one double-sided page of notes, but no other materials.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- To ensure fairness (and the appearance thereof), **proctors will not answer questions about the content of the exam**. If you are unsure of how to interpret a problem description, state your interpretation clearly and concisely. *Reasonable interpretations* will be taken into account by graders.

NAME: ____

CruzID: ___________________________ @ucsc.edu

Part I: Lambda calculus

1. **[16pts]** Use β-reductions to evaluate the following lambda term to a normal form.

(A) `((\p q -> p q) ((\x -> x) (\a b -> a))) (\k -> k)`

(B) `(\x y -> (y x) (\p q -> p)) (\i -> i) (\j -> j)`

2. **[8pts]** For each bound occurrence of a variable in the following lambda terms, draw an arrow pointing to its binder. For each free occurrence, draw a circle around the variable.

   (A) `(\a ->  b  (\b  a -> a  b))`

   (B) `(\p  q  r  -> (p  (\q  p  -> (r  q))) (q  p))`

3. **[16pts]** Fill in a lambda calculus expression for each blank in the program below to define a function `PROD` where `(PROD n)` returns the product of numbers between n and one. You may use any of the functions defined on the Lambda Calculus Cheat Sheet on the back page. Any other helper functions you must define yourself. Your implementation may assume that `PROD` is never called with `ZERO`.

```
let PROD1 = \f n -> ITE _____(A)_______
                        _____(B)_______
                        _____(C)_______

let PROD  = ______(D)_______
```

   (A)

   (B)

   (C)

   (D)

Part II: Haskell

4. **[5pts]** What does the following Haskell program evaluate to?

```
let f = (\x -> \y -> x + y)
    g = f 5
    h = \f n -> f (f n)
in
  h g 3
```

   (a) Type Error
   (b) `8`
   (c) `13`
   (d) `\f -> f (f 8)`
   (e) `16`

5. **[5pts]** What is the most general type of the Haskell function `foo`?

```
foo bar (x, y)
  | bar x       =  y ++ y
  | otherwise   =  y
```

   (a) `(a -> b) -> (a,b) -> [b]`
   (b) `(String -> Bool) -> (String, String) -> String`
   (c) `(a -> Bool) -> (a, a) -> [a]`
   (d) `(Bool -> a) -> [b] -> [b]`
   (e) `(a -> Bool) -> (a, [b]) -> [b]`

For the following questions, consider the data types defined below.

```
data TrickOrTreat = Trick Prank | Treat Candy

data Prank = Prank { desc :: String, legal :: YNM }

data YNM = Yes | No | Maybe

data Candy = Candy { pieces :: Int, kind :: Kind, rating :: Int}

data Kind = Chocolate | HardCandy | Gummies
```

6. **[21pts]** A `case` expression is *exhaustive* if all possible values are matched by at least one pattern. A pattern is *overlapping* if previous patterns match all values it matches. Assume `t` has type `TrickOrTreat` and do the following:

   - For each pattern in each `case`, provide a value that matches on the pattern.
   - If the pattern is overlapped by previous patterns, write "N/A".
   - Determine if the `case` expression is exhaustive and circle `Exhaustive` or `Non-exhaustive` as appropriate.
   - For non-exhaustive `case` expressions, write a value that does not match any of its patterns.

   (a)

```
                              case t of

_________________________       Trick (Prank _ Yes) -> ()

_________________________       Trick (Prank d No) -> ()

_________________________       Trick _ -> ()

_________________________       _ -> ()

_________________________     [ Exhaustive  /  Non-exhaustive ]
```

(b)

```
                              case t of
________________________        Treat (Candy _ k r) | r > 3 -> ()
________________________        Trick (Prank d l) -> ()
________________________        Treat c | (rating c) < 3 -> ()
________________________      [ Exhaustive  /  Non-exhaustive ]
```

(c)

```
                              case t of
________________________        Treat c -> ()
________________________        Treat (Candy n Chocolate r) -> ()
________________________        Trick p -> ()
________________________      [ Exhaustive  /  Non-exhaustive ]
```

(d)

```
                          case t of
_____________________       Trick (Prank "snakes on plane" No) -> ()
_____________________       Treat (Candy 99 Gummies r) -> ()
_____________________       Treat _ -> ()
_____________________     [ Exhaustive  /  Non-exhaustive ]
```

7. Consider a binary search tree where each internal node has a value i and two child subtrees. The left child contains all nodes with values less than or equal to i, and the right child contains all nodes with values strictly greater than i.

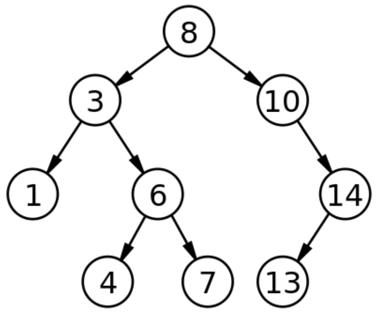

Figure 1: A binary search tree

(A) **[5pts]** Using the following ADT, create a `Tree` that represents the binary search tree in Figure 1.

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

(B) **[10pts]** Define the function `max`, which returns the highest value in a binary search tree represented by a `Tree` value, or 0 if the tree is empty. For example, if `t` is the tree in Figure 1, then `max t` evaluates to 14. Full credit requires an efficient traversal of the tree (e.g., not an exhaustive one). You may define helper functions if desired, but you may not use any library functions except `==`, `>=`, or `<=`.

```
max :: Tree -> Int
```

(C) **[10pts]** Define the function `contains`, which returns `True` if a number `n` is contained in a binary search tree `t`. For example, if `t` is the tree in Figure 1, then `contains 7 t` returns `True`, but `contains 5 t` returns `False`. Full credit requires an efficient traversal of the tree (e.g., not an exhaustive one). You may define helper functions if desired, but you may not use any library functions except `==`, `>=`, or `<=`.

```
contains :: Int -> Tree -> Bool
```

1 Lambda calculus cheat sheet

```
-- Booleans --------------------------------
let TRUE =\x y -> x
let FALSE = \x y -> y
let ITE = \b x y -> b x y
let NOT = \b x y -> b y x
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR = \b1 b2 -> ITE b1 TRUE b2

-- Numbers ---------------------------------
let ZERO = \f x-> x
let ONE = \f x -> f x
let TWO = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR = \f x -> f (f (f (f x))
let FIVE = \f x -> f (f (f (f (f x))

-- Arithmetic ------------------------------
let INC   = \n f x -> f (n f x)
let ADD   = \n m -> n INC m
let MUL   = \n m -> n (ADD m) ZERO
let ISZ   = \n ->   -- return TRUE if n == 0 --
let DECR  = \n ->   -- decrement n by one --
let EQL   = \a b -> -- return TRUE if a == b, otherwise FALSE --

-- Recursion -------------------------------
let FIX = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```