CSE 116, Fall 2019 Final

Section	Points	Score
Part I	35 points	
Part II	36 points	
Part III	97 points	
Total	168 points	

Instructions

- **You have 180 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. **[5pts]** What are the free variables of the lambda calculus expression
`(\a -> a (\b -> c) (\d e -> a d)) (\h i -> g)`

   (a) b, e, h, i

   (b) a, b, d, e, h, i

   (c) a, c, d, g

   (d) a, c, g

   (e) c, g

   (f) None of the above

   ANSWER E: c, g

2. **[14pts]** Use β-reductions to evaluate the following lambda terms to a normal form.

   (A) `(\x y -> x y (\a b -> a)) (\c d -> c) (\e f -> f)` Rubric:
   - 0 pts : no attempt or nothing correct.
   - 1 pts : anything correct (e.g., one reduction)
   - 2-5 pts : more than one thing correct (e.g., two reductions), few things incorrect
   - 6 pts : almost correct, but one smallish error
   - 7 pts : completely correct

   (B) `(\z x y -> x y (z y)) (\a -> a (\b c -> c) (\d e -> d)) (\f g -> g) (\h i -> i)`
   - `(\x y -> x y ((\a -> a (\b c -> c) (\d e -> d)) y)) (\f g -> g) (\h i -> i)`
   - `(\y -> (\f g -> g) y ((\a -> a (\b c -> c) (\d e -> d)) y)) (\h i -> i)`
   - `((\f g -> g) (\h i -> i) ((\a -> a (\b c -> c) (\d e -> d)) (\h i -> i)))`
   - `((\g -> g) ((\a -> a (\b c -> c) (\d e -> d)) (\h i -> i)))`
   - `(\h i -> i) (\b c -> c) (\d e -> d)`
   - `(\i -> i) (\d e -> d)`
   - `(\d e -> d)`

   Rubric:
   - 0 pts : no attempt or nothing correct.
   - 1 pts : anything correct (e.g., one reduction)
   - 2-5 pts : more than one thing correct (e.g., two reductions), few things incorrect
   - 6 pts : almost correct, but one smallish error
   - 7 pts : completely correct

3. **[16pts]** We can represent lists in lambda calculus using `PAIR` and `FALSE`. For example, the Haskell list `[0, 1, 2]` can be represented in Lambda Calculus as

   ```
   PAIR ZERO (PAIR ONE (PAIR TWO FALSE))
   ```

   and functions like Haskell's `head` and `tail` can be implemented by `FST` and `SND`, respectively. In the Haskell cheat sheet, you will also find `EMPTY`, which returns `TRUE` if applied to an empty list

   ```
   EMPTY FALSE =~> TRUE
   ```

   and returns `FALSE` otherwise:

   ```
   EMPTY (PAIR ZERO (PAIR ONE (PAIR TWO FALSE))) =~> FALSE
   ```

   Fill in a lambda calculus expression for each blank in the program below to define a function `FILTER` such that for a predicate function `p` and a list `l`, `(FILTER p l)` returns a list containing every element `v` of `l` such that `(p v) =~> TRUE`. You may use any of the functions defined on the Lambda Calculus Cheat Sheet on the back page.

   ```
   let FILTER1 = \f p l -> ITE (EMPTY l)
                               FALSE
                               ITE (p (FST l))
                                 (PAIR (FST l) (f p (SND l)))
                                 (f p (SND l))

   let FILTER  = FIX FILTER1
   ```

   (A)

   (B)

   (C)

   (D)

   (E)

   (F)

Part II: Nano

Consider the following grammar and operational semantics, and type system for Nano1.

Grammar

```
e ::= v
    | e1 e2

v ::= x | n
    | \x -> e
```

Operational semantics

```
           e1 => e1'
[App-L] ----------------
        e1 e2 => e1' e2

          e => e'
[App-R] ------------
        v e => v e'

[App]   (\x -> e) v => e[x := v]

               e1 => e1'
[Add-L]   --------------------
          e1 + e2 => e1' + e2

              e2 => e2'
[Add-R]   --------------------
          n1 + e2 => n1 + e2'

[Add]     n1 + n2 => n       where n == n1 + n2

                        e1 => e1'
[Let-Def] ------------------------------------
          let x = e1 in e2 => let x = e1' in e2

[Let]     let x = v in e2 => e2[x := v]
```

4. **[21pts]** Below are partial proofs that the expression `(\x -> \y -> x + y) (1+1) 6` evaluates to `8`. For each blank, fill in a typing rule or expression to complete the proof.

- `[Add]` `--------`
  `1+1 => 2`

  `[__a__]` `-----------------------------------------------`
  `(\x -> \y -> x + y) (1+1) => (\x -> \y -> x + y) 2`

  `[__b__]` `-------------------------------------------------------`
  `(\x -> \y -> x + y) (1+1) 6 => (\x -> \y -> x + y) 2 6`

- `[__c__]` `--------------------------------`
  `(\x -> \y -> x + y) 2 6 => __d__`

- `[__e__]` `--------------`
  `__f__ => 2 + 6`

- `[__g__]` `----------`
  `2 + 6 => 8`

(A) App-R
(B) App-L
(C) App
(D) `(\y -> 2 + y) 6`
(E) App
(F) `(\y -> 2 + y) 6`
(G) Add

5. **[5pts]** What is the most general unifier of the following two types (where `a,b,c` are type variables)?

```
(a -> b) -> b -> [a] -> b
(c -> Bool) -> a -> [c] -> [c]
```

(a) `[a / Bool, b / Bool, c / Bool]`

(b) `[a / Bool, b / [Bool], c / Bool]`

(c) `[a / [Bool], b / [Bool], c / Bool]`

(d) `[a / c, b / [c]]`

(e) None of the above

(f) Cannot unify

6. **[5pts]** What is the most general unifier of the following two types (where `a,b,c` are type variables)?

```
(a -> b) -> [a] -> b
(c -> b) -> [c] -> [c]
```

(a) `[b / c, a / b, c / b]`

(b) `[a / Bool, b / [Bool], c / Bool]`

(c) `[a / [Bool], b / [Bool], c / Bool]`

(d) `[a / c, b / [c]]`

(e) None of the above

(f) Cannot unify

7. **[5pts]** What is the result of applying the following substitution?

`[a / f, b / d, d / (e,e), f / (e -> c)] forall a. forall b. (d -> e -> f) -> (a -> b)`

```
forall a. forall b. ((e,e) -> e -> (e -> c)) -> (a -> b)
```

Part III: Haskell

8. **[5pts]** Given the following definition of `foo`:

```
foo a b =
    let c = \a -> b ++ a in
      let b = a ++ (c a) in
       c (bar b)
  where
    bar e = e ++ a ++ b
```

what does the expression `foo "x" "o"` evaluate to?

(a) `"xxxooo"`
(b) `"oxoxxo"`
(c) `"xxooxo"`
(d) `"xoxoxo"`
(e) `"oxoxox"`
(f) None of the above

9. **[5pts]** What does this Haskell expression evaluate to? (See Haskell cheat sheet for definition of map.)

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

(a) Type error
(b) `[0,1,2,3,4,5]`
(c) `[[0],[1],[2],[3],[4],[5]]`
(d) `[[0,1],[2,3],[4,5]]`
(e) `[[0,1,2,3,4,5]]`
(f) None of the above

10. **[5pts]** What does this Haskell expression evaluate to? (See Haskell cheat sheet for definition of foldl.)

```
foldl (:) [] ["a", "b", "c", "d"]
```

(a) Type error
(b) `"abcd"`
(c) `"dcba"`
(d) `["a","b", "c", "d"]`
(e) `["d","c", "b", "a"]`
(f) None of the above

11. **[5pts]** What does this Haskell expression evaluate to? (See Haskell cheat sheet for definition of filter.)

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

(a) Type error
(b) `(6,8)`
(c) `(2,4,3,5)`
(d) `[6, 8]`
(e) `[(2,4), (3,5)]`
(f) None of the above

Answer: E

12. **[5pts]** Recall that `(.)` is the infix operator for `compose`, so `(f.g) x` is the same as writing `f (g x)`. What does the following Haskell program evaluate to? (See the Haskell cheat sheet for types and definitions of the functions used below.)

```
(foldr (+) 0 . map (\x -> x*2) . filter even) [1,2,3,4,5]
```

(a) Type error
(b) 30
(c) 24
(d) 12
(e) 20
(f) None of the above

Answer: D

13. **[5pts]** What is the most general type of the Haskell function `foo`?

```
foo x y z = bar x y (z+1)
  where
    bar = \w u -> u w
```

(a) `Num c => a -> b -> c -> a`
(b) `Num c => (b -> c -> a) -> b -> c -> a`
(c) `Num b => a -> (a -> b -> c) -> b -> c`
(d) `Num b => (a -> b -> c -> d) -> a -> b -> c -> d`
(e) Type error
(f) None of the above

Answer: C

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

```
data Binop = Add | Sub | Mul

data Expr = Num Int
          | Var String
          | Bin Binop Expr Expr
          | Let String Expr Expr
```

14. **[15pts]** 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 `Expr` and do the following:

    - For each pattern in each `case`, provide a value that matches on the pattern.
    - If the pattern is overlapped by a previous pattern, write "overlapped".
    - 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.

    For example, the following patterns are **non-exhaustive**:

```
                                case t of

_Num 4___________                 Num n    -> ()

_Var "x"_________________         Var x -> ()

_overlapped_____________          Var _ -> ()

_Bin Mul (Var "x") (Num 1)_       Bin _ x y -> ()

_Let "x" (Num 1) (Num 2)__        Let id _ (Num y) -> ()

_Let "x" (Num 1) (Var "y")__      [ Exhaustive  /  (( Non-exhaustive )) ]
```

(a)

```
case t of

  Var foo -> ()

  Bin op x y  -> ()

  Let x e1 e2 -> ()

  Num n -> ()

  _ -> ()

[ Exhaustive  /  Non-exhaustive ]
```

(b)

```
case t of

  Num 1 -> ()

  _ -> ()

[ Exhaustive  /  Non-exhaustive ]
```

(c)

```
case t of

_____________________      Bin Add (Var x) (Var y)
                             | x ++ y == "foobar" -> ()

_____________________      Bin Add x y -> ()

_____________________      Let _ _ _ -> ()

_____________________      Var x -> ()

_____________________      Num x -> ()

_____________________    [ Exhaustive  /  Non-exhaustive ]
```

Consider the datatype below for a tree where each internal node has a value (of type `a`) and two child subtrees, eventually terminating with a leaf node.

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

15. **[2pts]** What is the most general type of the value `t`?

```
t = (Node "foo"
      (Node "bar" (Node "baz" Leaf Leaf) Leaf)
      (Node "qux" Leaf (Node "quux" Leaf Leaf)))
```

`Tree String` or `Tree Char`

16. **[10pts]** Implement a fold for the `Tree` datatype so that `foldTree (++) "" t` evaluates to `""foobarbazquxquux"`. You may only use library functions that appear in the Haskell cheat sheet or elsewhere in the exam.

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

foldTree f d Leaf = d
foldTree f d (Node v l r) = f v (foldTree f (foldTree f d r) l)
```

17. **[5pts]** Implement `maxIntTree`, which returns the largest `Int` in a tree of integers (or 0 for an empty tree). You may only use library functions that appear in the Haskell cheat sheet or elsewhere in the exam.

```
maxIntTree :: Tree Int -> Int

maxIntTree = foldTree max 0
```

18. **[5pts]** Now implement `maxTree`, a version of `maxIntTree` that works for any kind of tree. That is, for a tree of type `Tree a`, it returns the maximum value or a default value for the empty tree. You may only use library functions that appear in the Haskell cheat sheet or elsewhere in the exam. You must also specify the type of your implementation. Note that the tree elements may appear in any order—do not assume the tree is a BST.

```
maxTree :: Ord a => a -> Tree a -> a
maxTree d = foldTree max d
```

19. **[10pts]** Implement `joinTree` which creates a single tree out of two trees by replacing the left-most leaf node of the second tree.

For example, the output of `joinTree (Node "boo" Leaf Leaf) t` should return

```
(Node "foo"
  (Node "bar" (Node "baz"(Node "boo" Leaf Leaf) Leaf) Leaf)
  (Node "qux" Leaf (Node "quux" Leaf Leaf)))
```

```
joinTree :: Tree a -> Tree a -> Tree a

joinTree t Leaf = t
joinTree t (Node v l r) = Node v (joinTree t l) r
```

20. **[20pts]** Implement a `filter` function for the `Tree` datatype which takes a predicate and a tree and returns a tree containing only the values for which the predicate was true. For example, `filterTree (\x -> length x >= 4) t` should return

```
(Node "quux" Leaf Leaf)
```

and `filterTree (\x -> x /= "foo") t` should return

```
(Node "qux" (Node "bar" (Node "baz" Leaf Leaf) Leaf)
            (Node "quux" Leaf Leaf))
```

You may only use library functions that appear in the Haskell cheat sheet or elsewhere in the exam.

```
filterTree :: (a -> Bool) -> Tree a -> Tree a

filterTree p Leaf = Leaf
filterTree p (Node v l r) | p v =
  Node v (filterTree p l) (filterTree p r)
filterTree p (Node v l r) | otherwise =
  case filterTree p l of
    Leaf -> filterTree p r
    fl   -> case filterTree p r of
              Leaf -> fl
              fr   -> joinTree fl fr
```

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))

-- Pairs -----------------------------------
let PAIR = \x y b -> b x y
let FST = \p -> p TRUE
let SND = \p -> p FALSE

-- Lists -----------------------------------
let EMPTY = \xs -> xs (\x y z -> FALSE) TRUE
let CONS  = PAIR
let NIL   = FALSE
let HEAD  = FST
let TAIL  = SND

-- 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))
```

2 Haskell cheat sheet

```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
```