CMPS 112, Spring 2019 Final

Section	**Points**	**Score**
Part I	50 points	
Part II	34 points	
Part III	50 points	
Total	134 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: General multiple choice

Select **one** answer that best answers each question.

1. **[3pts]** What are the free variables of the lambda calculus expression `(\x -> x y (\z -> x)) (\a b -> x)`?
   (a) x, y, z
   (b) x, z, a, b
   (c) x, y
   (d) x
   (e) y
2. **[3pts]** `(\x -> (\y -> x)) apple =b> ???`
   (a) `apple`
   (b) `\y -> apple`
   (c) `\x -> apple`
   (d) `\y -> y`
   (e) `\x -> y`
3. **[3pts]** Which of the following lambda calculus terms are in normal form ?
   (a) `(\x -> x x) y`
   (b) `x y`
   (c) `(\x -> x x) (\y -> y y)`
   (d) `x (\y -> y)`
   (e) A and C
   (f) B and D
4. **[4pts]** Which of the following is **not** a pattern in Haskell?
   (a) `((1,_):xs)`
   (b) `x:[]`
   (c) `[x]`
   (d) `[x,y,[z]]`
   (e) all of the above are patterns

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

```
foo bar (x:xs)
  | bar x       = x : foo bar xs
foo bar (x:xs) = foo bar xs
foo bar []     = []
```

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

(b) `(Int -> Bool) -> [Int] -> [Bool]`

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

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

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

6. **[4pts]** What does the following Haskell program evaluate to?
(See Haskell cheat sheet for `map` definition)

```
let f = (\x -> \y -> x + y) in
  map (f 3) [1, 2, 3]
```

(a) Type Error

(b) `[9]`

(c) `[(1, 3), (2, 3), (3, 3)]`

(d) `[4, 5, 6]`

(e) `[1, 2, 3]`

7. **[5pts]** What does this Haskell program evaluate to?
(See Haskell cheat sheet for `foldl` definition)

```
foldl (++) "" ["foo", "bar", "baz"]
```

(a) Type error

(b) `["baz", "bar", "foo"]`

(c) `["foo", "bar", "baz"]`

(d) `"foobarbaz"`

(e) `"bazbarfoo"`

(f) `"zabraboof"`

8. **[5pts]** What does this Haskell program evaluate to?
(See Haskell cheat sheet for `foldr` definition)

```
foldr (++) "" ["foo", "bar", "baz"]
```

(a) Type error
(b) `["baz", "bar", "foo"]`
(c) `["foo", "bar", "baz"]`
(d) `"foobarbaz"`
(e) `"bazbarfoo"`
(f) `"zabraboof"`

9. **[4pts]** What does this Haskell expression evaluate to?

```
let a = "foo" in
 let b = "bar" in
  let c = "baz" in
   let f b = a ++ b ++ c in
    let b = "qux" in
     let c = "fred" in
      f b
```

(a) `"foobarbaz"`
(b) `"foobarfred"`
(c) `"fooquxfred"`
(d) `"fooquxbaz"`
(e) None of the above

10. **[6pts]** A `case` expression is *exhaustive* if all possible values are matched by at least one pattern. Consider the following data type:

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

Assuming `p` has type `Paragraph`, which of the following case statements are **not** exhaustive?

(a)
```
case p of
  Heading n str -> ...
  Text str -> ...
  List _ els -> ...
```

(b)
```
case p of
  Heading n _ -> ...
  List b [_] -> ...
  Text str -> ...
```

```
case p of
  _ -> ...
```

(d)
```
case p of
  Text _ -> ...
  List _ _ -> ...
  Heading _ _ -> ...
```

(e)
```
case p of
  _ ->  ...
  List b [_] -> ...
```

(f) They are all exhaustive

11. **[3pts]** What does the following Haskell program evaluate to?

```
case (Heading 3 "Intro") of
  Heading n str -> "foo"
  Heading 3 str -> "bar"
  Heading _ str -> "baz"
  _             -> "qux"
```

(a) "foo"

(b) "bar"

(c) "baz"

(d) "qux"

(e) Type error

12. **[5pts]** What does the following Haskell program evaluate to?
(See Haskell cheat sheet for `filter` definition)

```
let ps = [Heading 3 "head", Text "text", List True ["item1"]] in
let f = (\p -> case p of
                List b item -> True
                _ -> False) in
  filter f ps
```

(a) [False, False, True]

(b) "headtext"

(c) [Heading 3 "head", Text "text"]

(d) [List True ["item1"]]

(e) "item1"

(f) None of the above

Part II: Syntax and typing

13. **[4pts]** Consider the following grammar for lambda calculus, where `x` is any token matched by the regular expression `[a-zA-Z][a-zA-Z0-9]*`.

```
e := \x -> e
   | e1 e2
   | x
   | ( e )
```

According to this grammar, which of the following expressions are syntactically **invalid**?

(a) `\(x9 yy) -> yy`

(b) `\y -> x0y`

(c) `x x`

(d) `(e (e))`

(e) all of the above are invalid

(f) all of the above are valid

14. **[3pts]** Consider the partially implemented algebraic data type below that represents abstract syntax trees (ASTs) for the grammar above.

```
data Expr =
   | Abs __a__ __b__
   | App __c__ Expr
   | Var String
```

For each blank (labeled `a-c`), fill in a type to complete the data type.

(a) ___________________________________

(b) ___________________________________

(c) ___________________________________

15. **[3pts]** What is a possible AST representing the expression

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

(a) `(Abs "x" "y" (App "x" "y")) (Var "z")`

(b) `(Abs "x" (Abs "y" (App "x" "y"))) (Var "z")`

(c) `(App (Abs "x" (Abs "y" (App (Var "x") (Var "y")))) (Var "z"))`

(d) `(App (Abs (Var "x") (Abs (Var "y") (App (Var "x") (Var "y")))) (Var "z"))`

(e) none of the above are valid

16. **[12pts]** Consider the following type system for lambda calculus. (Note: this is a subset of the type system we used for Nano.)

```
[T-Var] -------------------
        [x:T] |- x :: T

          G, x:T1 |- e :: T2
[T-Abs] ------------------------
        G |- \x -> e :: T1 -> T2

        G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App] -----------------------------------
                 G |- e1 e2 :: T2
```

Types are represented by the following grammar:

```
T  := Int
    | T1  -> T2
```

Below is a partial proof that the expression `(\x -> \y -> y) z` is well-typed in the typing context `G = [z:Int]`. For each blank (labeled `a-f`), fill in a typing rule, expression, or type to complete the proof.

```
[__a__] ---------------------------
        [z:Int,x:Int,y:Int] |- y :: Int
[__b__] ----------------------------------
        [z:Int,x:Int] |- __c__ :: Int -> Int
[T-Abs] ---------------------------------------  -----------------------[T-Var]
        [z:Int,x:Int] |- \x -> \y -> y :: __d__  [z:Int] |- __e__ :: Int
[__f__] -------------------------------------------------------------
        [z:Int] |- (\x -> \y -> y) z :: Int -> Int
```

(a) ___________________________________

(b) ___________________________________

(c) ___________________________________

(d) ___________________________________

(e) ___________________________________

(f) ___________________________________

17. **[4pts]** What is the unifier of the following two types (where `a,b,x,y` are type variables)?

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

(a) `[a / x, b / Int, y / Int]`
(b) `[x / a, b / Int, y / Int]`
(c) `[x / a, b / (y -> Int)]`
(d) `[x / (a -> Int), b / (y -> Int)]`
(e) A or B
(f) None of the above
(g) Cannot unify

18. **[4pts]** What is the result of applying the following substitution?

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

(a) `(a -> a) -> (a -> a) - f`
(b) `forall a . (a -> a) -> (a -> a) - f`
(c) `forall c . (a -> a) -> (c -> c) - f`
(d) `forall a . b -> (a -> a) - f`
(e) None of the above

19. **[4pts]** Which of the following types is a valid instantiation of the polymorphic type `forall a . forall b . (a -> b) -> [(a, c)] -> [b]`

(a) `forall b . (c -> b) -> [(c, c)] -> [b]`
(b) `(d -> e) -> [(d, f)] -> [e]`
(c) `(c -> d) -> [(c, c)] -> [d]`
(d) `(d -> e) -> [(f, g)] -> [h]`
(e) None of the above

Part III: Recursion and folding

Given a list of numbers, a number `old`, and a number `new`, `replace` returns a list of numbers where every occurrence of `old` has been replaced by `new`

```
replace :: [Int] -> Int -> Int -> [Int]
```

Your implementations must pass the following test cases.

```
replace [1, 2, 3] 2 4
  ==>[1, 4, 3]

replace [1, 2, 3] 4 5
  ==>[1, 2, 3]

replace [1, 2, 2] 2 4
  ==>[1, 4, 4]
```

Unless noted, you may only use the following library functions. (You may also use the list constructors `(:)` and `[]`.)

```
(==) :: Eq a => a -> a -> Bool

(++) :: [a] -> [a] -> [a]

reverse :: [a] -> [a]
```

20. **[10pts] Head recursive `replace`** Implement the Haskell function `replace` **using head recursion**.

```
replace :: [Int] -> Int -> Int -> [Int]
```

21. **[10pts] Tail recursive `replace`** Implement the Haskell function `replace` **using tail recursion**.

```
replace :: [Int] -> Int -> Int -> [Int]
```

22. **[10pts] Left fold `replace`**

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Implement the Haskell function `replace` using `foldr` (in addition to any of the permitted library functions). **Your implementation should not contain any recursive calls**.

```
replace :: [Int] -> Int -> Int -> [Int]
```

23. **[10pts] Right fold `replace`**

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Implement the Haskell function `replace` using `foldr` (in addition to any of the permitted library functions). **Your implementation should not contain any recursive calls**.

```
replace :: [Int] -> Int -> Int -> [Int]
```

24. **[10pts] Lambda calculus `replace`**
Now write a lambda calculus function `replace` for Church-encoded natural numbers. You may use any function in the "Lambda Calculus cheat sheet" as well as the equality function `EQL`. `EQL n m` returns `TRUE` if `n` and `m` represent the same number, and `FALSE` otherwise. Your function may use head or tail recursion.

Your implementations must pass the following test cases.

```
eval test0:
  REPLACE FALSE TWO FOUR
    =~> FALSE

eval test1:
  REPLACE (PAIR ONE (PAIR TWO (PAIR THREE FALSE))) TWO FOUR
    =~> (PAIR ONE (PAIR FOUR (PAIR THREE FALSE)))

eval test2:
  REPLACE (PAIR ONE (PAIR TWO (PAIR THREE FALSE))) FOUR FIVE
    =~> (PAIR ONE (PAIR TWO (PAIR THREE FALSE)))

eval test3:
  REPLACE (PAIR ONE (PAIR TWO (PAIR TWO FALSE))) TWO FOUR
    =~> (PAIR ONE (PAIR FOUR (PAIR FOUR FALSE)))
```

```
let REPLACE =
```

Reference material (You may detach this sheet)

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

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

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

-- Lists ----------------------------------
let EMPTY = \xs -> xs (\x y z -> FALSE) TRUE
let APPEND = FIX (\rec l1 l2 ->
                   ITE (EMPTY l1)
                     l2
                     (PAIR (FST l1) (rec (SND l1) l2)))

-- 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 -> n (\z -> FALSE) TRUE
```

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
```