CSE114A, Winter 2025: Final Exam

Instructor: Lindsey Kuper

March 17, 2025

Student name: ____

CruzID: ____@ucsc.edu

This exam has 19 questions and 100 total points.

Instructions

- Please write directly on the exam.
- For short answer questions, please write your answer in the provided boxes. You can use space outside of the boxes as scratch space, but we won't see or grade it.
- For multiple choice questions, please circle the correct choice.
- **You have 180 minutes to complete this exam.** You may leave when you are finished.
- 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.
- Please, no talking. No notes, books, laptops, phones, or other electronic devices. 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.
- **We will give partial credit for partially correct answers when it makes sense to do so.** A partially correct answer is better than leaving an answer blank.

Good luck!

This page is for your use as scratch space. Anything you write here will be ungraded.

Part 1: Lambda Calculus

1. (3 points) A lambda calculus expression is in *normal form* if it cannot be further reduced. Evaluate the following lambda calculus expression to normal form using a series of β-reduction steps (and only β-reduction steps – you shouldn't need anything else). Start each line with `=b>`, as if you were using Elsa, and do just one β-reduction step per line.

```
(\x y -> y x y) (\q r -> q) (\f z -> f (f z))
```

Solution: Here is one possible solution:

```
 (\x y -> y x y) (\q r -> q) (\f z -> f (f z))
=b> (\y -> y (\q r -> q) y) (\f z -> f (f z))
=b> (\f z -> f (f z)) (\q r -> q) (\f z -> f (f z))
=b> (\z -> (\q r -> q) ((\q r -> q) z)) (\f z -> f (f z))
=b> (\q r -> q) ((\q r -> q) (\f z -> f (f z)))
=b> \r -> ((\q r -> q) (\f z -> f (f z)))
=b> \r -> (\r -> (\f z -> f (f z)))
```

And here's another:

```
 (\x y -> y x y) (\q r -> q) (\f z -> f (f z))
=b> (\y -> y (\q r -> q) y) (\f z -> f (f z))
=b> (\f z -> f (f z)) (\q r -> q) (\f z -> f (f z))
=b> (\z -> (\q r -> q) ((\q r -> q) z)) (\f z -> f (f z))
=b> (\z -> (\q r -> q) (\r -> z)) (\f z -> f (f z))
=b> (\q r -> q) (\r -> (\f z -> f (f z)))
=b> \r -> (\r -> (\f z -> f (f z)))
```

Any correct reduction sequence will end in `\r -> (\r -> (\f z -> f (f z)))`.

2. (3 points) Which of the following is true?

   ◯ There *may* be **more than one** sequence of reduction steps that reduces a given lambda calculus expression to normal form, **but for the particular expression from question 1**, there is **exactly one**.

   √ There *may* be **more than one** sequence of reduction steps that reduces a given lambda calculus expression to normal form, **and for the particular expression from question 1**, there is **more than one**.

   ◯ There is *always* **exactly one** sequence of reduction steps that reduces a given lambda calculus expression to normal form.

   ◯ There is *always* **more than one** sequence of reduction steps that reduces a given lambda calculus expression to normal form.

For the next two questions, you may use any of the helper functions defined in the provided lambda calculus reference at the end of the exam.

Recall from the midterm that we can encode *lists* in lambda calculus using pairs, as follows:

```
let NIL   = \x -> TRUE
let CONS  = PAIR
let HEAD  = FST
let TAIL  = SND
let ISNIL = \lst -> lst (\h t -> FALSE)
```

The list constructors are `NIL` and `CONS`. `NIL` is the empty list, and a non-empty list consists of a pair of a list element and a list.

For instance, `(CONS ONE (CONS TWO (CONS THREE NIL)))` is analogous to the Haskell list `[1, 2, 3]`, which is just syntactic sugar for `1 : (2 : (3 : []))`.

The `ISNIL` function takes a list and returns `TRUE` if the list is `NIL` and `FALSE` otherwise.

(Note: Unlike in Haskell, elements of these lists don't all have to be of the same type! It's fine to have a list like `CONS FALSE (CONS ONE NIL)`, for example.)

3. (2 points) Define a lambda calculus function `PREPENDFALSE` that takes a list and returns a new list that has `FALSE` as its first element, followed by all the elements of the original list. The resulting list is therefore one element longer than the original list was.

   You may assume that `PREPENDFALSE` is called with a list constructed with `NIL` or `CONS`. Here are some example calls to `PREPENDFALSE`:

```
PREPENDFALSE NIL =*> CONS FALSE NIL
PREPENDFALSE (CONS TRUE NIL) =*> CONS FALSE (CONS TRUE NIL)
PREPENDFALSE (CONS ONE NIL) =*> CONS FALSE (CONS ONE NIL)

let PREPENDFALSE =
```

Solution: One simple correct answer is:

```
\lst -> CONS FALSE lst
```

4. Define a lambda calculus function `SUMLIST` that takes a list of Church numerals and returns the sum of its elements as a Church numeral.

   Here are some example calls to `SUMLIST`:

```
SUMLIST NIL =~> ZERO
SUMLIST (CONS TWO NIL) =~> TWO
SUMLIST (CONS TWO (CONS ONE NIL)) =~> THREE
SUMLIST (CONS ZERO (CONS THREE (CONS ZERO NIL))) =~> THREE
```

   You may assume that `SUMLIST` is called with a list constructed with `NIL` or `CONS`, and that all of the list's elements are Church numerals. You must use recursion for full credit.

```
let SUMLIST1 = \rec -> \lst -> ITE ______(part 4(a))________
                                   ______(part 4(b))________
                                   ______(part 4(c))________

let SUMLIST = _______(part 4(d))________
```

   a. (3 points) 4(a):

      Solution: This is where we check a condition to know whether we are in the base case or not. `(ISNIL lst)` is a correct answer.

   b. (3 points) 4(b):

      Solution: This is the base case. `ZERO` is a correct answer.

   c. (3 points) 4(c):

      Solution: This is the recursive case. `(ADD (HEAD lst) (rec (TAIL lst)))` is a correct answer.

   d. (3 points) 4(d):

      Solution:

      `Y SUMLIST1`

Part 2: Haskell

The Haskell reference at the end of the exam has information about library functions used in this section.

5. (3 points) What is the **type** of the following Haskell expression?

```
\q r s -> [q, r]
```

◯ Type error
◯ `a -> a -> a -> [b]`
◯ `a -> a -> b -> [b]`
◯ `a -> a -> a -> [a]`
√ `a -> a -> b -> [a]`

6. (3 points) What is the **type** of the following Haskell expression?

```
\q r s -> [q, r, r == s]
```

◯ Type error
√ `Bool -> Bool -> Bool -> [Bool]`
◯ `Eq a => Bool -> a -> a -> [Bool]`
◯ `Eq a => a -> a -> a -> [a]`
◯ `(Eq a, b) => a -> b -> b -> [a]`

7. (3 points) What is the **type** of the following Haskell expression?

```
\x y -> (x, y, "batman", "catman", "wingman")
```

◯ Type error
◯ `a -> a -> (a, a, String, String, String)`
√ `a -> b -> (a, b, String, String, String)`
◯ `a -> a -> [String]`
◯ `a -> b -> [String]`

8. (3 points) What is the **type** of the following Haskell expression?

```
\y z -> map (\x -> "kona" ++ x) [y, z]
```

◯ Type error
◯ `[String] -> [String] -> String`
◯ `[String] -> [String] -> [String]`
√ `String -> String -> [String]`
◯ `[String] -> [String] -> [String]`

9. For this problem, you can use any library functions from the Haskell reference at the end of the exam, but no other library functions.

   Consider a Haskell function `andList :: [Bool] -> Bool` that takes a list of expressions of `Bool` type and returns `True` if all elements of the list evaluate to `True`, and `False` otherwise. Here are some example calls to `andList`:

```
ghci> andList []
True
ghci> andList [True, False, True, True]
False
ghci> andList [True, 3 == 3, True && True]
True
```

   a. (3 points) Define `andList` in the box below. The type signature is provided for you. Your definition should use pattern matching, and should *not* be tail-recursive.

   `andList :: [Bool] -> Bool`

   Solution: One simple correct solution is:

```
andList []     = True
andList (x:xs) = x && andList xs
```

   Since the question calls for a definition that uses pattern matching, a definition that uses `head` and `tail` would not be correct. (Those functions aren't in the allowed library functions.)

   b. (3 points) Now define `andList'`, which has the same type signature and behavior as `andList`, but is written using `foldr`. (Hint: You can do this in one line of code.)

   `andList' :: [Bool] -> Bool`

   Solution:

```
andList' l = foldr (&&) True l
```

   It's also OK to eta reduce:

```
andList' = foldr (&&) True
```

   c. (4 points) Finally, define `andListTR`, which has the same type signature and behavior as `andList`, but is tail-recursive.

   `andListTR :: [Bool] -> Bool`

   Solution: One approach is to use a helper function:

```
andListTR l = helper l True
  where helper :: [Bool] -> Bool -> Bool
        helper []     b = b
        helper (x:xs) b = helper xs (x && b)
```

   You do not have to give a type signature to the helper function, though it might make it easier to read. (It's also possible to do this with `foldl`, but `foldl` is not one of the allowed library functions.)

Part 3: Working with Abstract Syntax Trees

In this section and the next section, we'll use the following `Expr` data type, which defines a grammar of abstract syntax trees (ASTs) for *SmolHaskell*, a language with variables, integer literals, `let`-expressions, addition, function definitions, and function calls:

```
data Expr = Var String           -- Variable
          | Num Int              -- Integer literal
          | Let String Expr Expr -- let-expression: `let x = e1 in e2`
          | Add Expr Expr        -- Addition: `e1 + e2`
          | Lam String Expr      -- Function definition
          | App Expr Expr        -- Function call
```

For example, the program `let f = \z -> z + 5 in f 3` has the following AST:

```
Let "f" (Lam "z" (Add (Var "z") (Num 5))) (App (Var "f") (Num 3))
```

10. (2 points) Be the parser: translate the following SmolHaskell program into its corresponding `Expr`.

```
(let x = 4 in x + 5) + ((\x -> x) 7)
```

Solution:

```
Add (Let "x" (Num 4) (Add (Var "x") (Num 5)))
    (App (Lam "x" (Var "x")) (Num 7))
```

11. (5 points) Let us define the *depth* of a SmolHaskell expression as follows:

- The depth of a variable or an integer literal is 1.
- The depth of a lambda abstraction `\x -> e` is 1 + the depth of `e`.
- The depth of a `let`-expression `let x = e1 in e2`, an addition expression `e1 + e2`, or an application `e1 e2` is 1 + the maximum of the depth of `e1` and the depth of `e2`.

Define a Haskell function `depth` that takes an `Expr` and returns its depth as an `Int`. You can use the library functions in the Haskell reference at the end of the exam, but no other library functions. The type signature of `depth` is provided for you below; fill in the rest of the definition. Here are some sample calls to `depth`:

```
> depth (Var "x")
1
> depth (Add (Var "x") (Num 4))
2
> depth (Add (Add (Var "x") (Num 1)) (Num 2))
3
> depth (Lam "x" (Add (Add (Var "x") (Num 1)) (Num 2)))
4
> depth (App (Lam "x" (Var "x")) (Lam "x" (Var "x")))
3
> depth (Let "x" (Num 5) (Add (Var "x") (Num 3)))
3
```

```
depth :: Expr -> Int
```

Solution: A simple implementation is:

```
depth :: Expr -> Int
depth (Var s) = 1
depth (Num n) = 1
depth (Let s e1 e2) = 1 + max (depth e1) (depth e2)
depth (Add e1 e2) = 1 + max (depth e1) (depth e2)
depth (Lam s e) = 1 + depth e
depth (App e1 e2) = 1 + max (depth e1) (depth e2)
```

12. (10 points) An occurrence of a variable in a SmolHaskell expression is *free* if it is not bound by an enclosing lambda abstraction or `let` binding.

For example, in the expression `let x = 5 in x + 3`, there are no free occurrences of variables; in the expression `let x = 5 in x + y`, there is a free occurrence of `y`; and in `(\x -> y) x`, both `y` and `x` occur free.

Define a Haskell function `freeVars` that takes an `Expr` and returns a list of variables that occur free in it (in any order). You can use the library functions in the Haskell reference at the end of the exam, but no other library functions. The type signature of `freeVars` is provided for you below; fill in the rest of the definition.

Here are some sample calls to `freeVars`:

```
-- x + y
> freeVars (Add (Var "x") (Var "y"))
["x","y"]
-- \y -> x + x
> freeVars (Lam "y" (Add (Var "x") (Var "x")))
["x"]
-- (let x = 5 in x) + (let y = 5 in x)
> freeVars (Add (Let "x" (Num 5) (Var "x")) (Let "y" (Num 5) (Var "x")))
["x"]
```

For full credit, a variable that occurs free more than once in an expression should only appear once in the list returned by `freeVars`. Hint: Use the `nub` and `(\\)` list operations.

```
freeVars :: Expr -> [String]
```

Solution: In a `let`-expression `let x = e1 in e2`, if the language supports recursive `let`, then occurrences of `x` in `e1` **are not** considered free. In that case, this is a correct solution:

```
freeVars :: Expr -> [String]
freeVars (Var s) = [s]
freeVars (Num _) = []
freeVars (Let s e1 e2) = nub (freeVars e1 ++ freeVars e2) \\ [s]
freeVars (Add e1 e2) = nub (freeVars e1 ++ freeVars e2)
freeVars (Lam s e) = freeVars e \\ [s]
freeVars (App e1 e2) = nub (freeVars e1 ++ freeVars e2)
```

On the other hand, if the language does **not** support recursive `let`, then occurrences of `x` in `e1` **are** considered free. In that case, the `Let` case of `freeVars` would change to:

```
freeVars (Let s e1 e2) = nub (freeVars e1 ++ (freeVars e2 \\ [s]))
```

We'll accept either answer for this problem.

Part 4: Interpreters and Environments

13. (20 points) Next, we'll be writing an environment-passing interpreter for `Expr`s, so let's set up some machinery for doing that. Ideally, an `Expr` will evaluate to a `Value`, as defined by the following data type:

```
data Value = ValNum Int | ValClos String Expr ListEnv
```

where `ListEnv` is a simple list representation of an environment, which maps program variables (represented as `String`s) to their `Value`s:

```
type ListEnv = [(String, Value)]
```

Here is the function that we'll use for looking up the values of program variables in an environment:

```
lookupInEnv :: ListEnv -> String -> Maybe Value
lookupInEnv [] k = Nothing
lookupInEnv ((k',v):env') k =
  if k == k' then Just v else lookupInEnv env' k
```

The type signature of our interpreter will be:

```
eval :: ListEnv -> Expr -> Maybe Value
```

The `Maybe` type is there because things can go wrong during the evaluation of an `Expr`. In particular, it could contain an unbound variable, like `Add (Var "x") (Num 5)`, or it could be ill-typed, like `App (Num 3) (Num 5)` or `Add (Lam "x" (Var "x")) (Num 5)`. If either of those things happen, we want our interpreter to return `Nothing`. If nothing goes wrong, our interpreter should return a `Value` wrapped in the `Just` constructor.

We are now ready to implement `eval`. The type signature and the cases for `Var` and `Add` expressions are provided for you; your job is to implement the `Num`, `Lam`, `App`, and `Let` cases.

Hints:

- Functions should evaluate to closure values.
- In a `let`-expression `let x = e1 in e2`, you will want to evaluate `e1`, then evaluate `e2` in an extended environment. Your interpreter does *not* need to support recursive functions.
- In an application expression `e1 e2`, you will want to evaluate `e1` to a closure, and then evaluate the closure body in an extended environment.
- Because we're using a list representation of environments, you can use `(:)` to add things to an environment.
- You can use `case` expressions to pattern match on the result of recursive calls. Look at how the `Add` case is written for an example.

```
eval :: ListEnv -> Expr -> Maybe Value
eval env (Var x) = lookupInEnv env x
eval env (Add e1 e2) = case (eval env e1, eval env e2) of
    (Just (ValNum v1), Just (ValNum v2)) -> Just (ValNum (v1 + v2))
    _                                    -> Nothing
```

Solution:

```
eval _   (Num n) = Just (ValNum n)
eval env (Lam s body) = Just (ValClos s body env)
eval env (Let s e1 e2) = case eval env e1 of
    Just v -> eval ((s,v):env) e2
    _      -> Nothing
eval env (App e1 e2) = case (eval env e1, eval env e2) of
    (Just (ValClos s b e), Just a) -> eval ((s,a):e) b
    _                              -> Nothing
```

There is no need to write an `extendEnv` helper function for this question (since it would just call `(:)`), but it is OK to do so.

Part 5: Unification and Type Inference

In the context of unification and type inference, we've discussed the concept of *substitutions*. A substitution is a mapping from *type variables* to *types*, where the types may themselves contain type variables (or be type variables). For example,

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

is a substitution that maps the type variable `a` to the type `Bool`, the type variable `b` to the type `Int -> c`, and the type variable `d` to the type `e`.

14. (2 points) If possible, write down a substitution that is *a unifier* for the types `Int -> a` and `a -> b`. If these types do not unify, write "Cannot unify".

Solution: One simple correct solution is:

```
[(a, Int), (b, Int)]
```

Other solutions are also possible.

15. (2 points) If possible, write down a substitution that is *a unifier* for the types `Int -> a` and `a`. If these types do not unify, write "Cannot unify".

Solution: Cannot unify

16. (2 points) If possible, write down a substitution that is *a unifier* for the types `a -> b` and `c -> Int -> d`. If these types do not unify, write "Cannot unify".

Solution: One correct solution is:

```
[(a, c), (b, Int -> d)]
```

Here, `c -> Int -> d` is syntactic sugar for `c -> (Int -> d)`, so `[(a, c -> Int), (b, d)]` would be an incorrect answer. However, other solutions are possible.

17. (2 points) If possible, write down a substitution that is *a unifier* for the types `Bool` and `a -> b`. If these types do not unify, write "Cannot unify".

Solution: Cannot unify

18. (2 points) If possible, write down a substitution that is *a unifier* for the types `a -> b` and `a -> c`. If these types do not unify, write "Cannot unify".

Solution: One correct solution is:

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

Other solutions are possible. For instance, `[(c, b)]` is also correct.

19. In section 4 of this exam, we wrote an interpreter for SmolHaskell. Here's what should happen if you tried to evaluate the expression `(\x -> x) + 3`:

```
> eval [] (Add (Lam "x" (Var "x")) (Num 3))
Nothing
```

To catch these kinds of errors prior to run time, we can define a *type system* for SmolHaskell and then implement a *type checker*. Below are the typing rules we'll use, all of which are standard:

```
              (x,T1):G |- e :: T2                     (x,T) in G
[T-Lam] --------------------------          [T-Var] -----------
        G |- (\x -> e) :: T1 -> T2                  G |- x :: T


        G |- e1 :: Int       G |- e2 :: Int
[T-Add] -----------------------------------
                 G |- e1 + e2 :: Int


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

        G |- e1 :: T1    (x,T1):G |- e2 :: T2
[T-Let] -------------------------------------    [T-Int] -------------
             G |- let x = e1 in e2 :: T2                 G |- n :: Int
```

Let's use our typing rules to make sure that an expression in SmolHaskell is well-typed. The expression we'll consider is `let f = \x -> x + 1 in f 2`.

For each blank below, fill in a type or the name of a typing rule to complete the typing derivation.

We are using the following abbreviations for type environments:

```
G1 = [(x,Int)]
G2 = [(f,Int -> Int)]
```

```
     (x,Int) in G1
[19a]------------- [19b]-----------
    G1 |- x::Int       G1 |- 1::Int       (f,Int->Int) in G2
[19c]------------------------------- [19d]------------------  [19e]--------------
             G1 |- x+1 :: [19f]         G2 |- f :: Int->Int      G2 |- 2 :: Int
       [19g]------------------------ [19h]---------------------------------------
            [] |- \x -> x+1 :: [19i]               G2 |- f 2 :: [19j]
       [19k]-----------------------------------------------------------
                  [] |- let f = \x -> x + 1 in f 2 :: Int
```

a. (1 point) 19(a):

Solution:
`T-Var`

b. (1 point) 19(b):

Solution:
`T-Int`

c. (1 point) 19(c):

Solution:
`T-Add`

d. (1 point) 19(d):

Solution:
`T-Var`

e. (1 point) 19(e):

Solution:
`T-Int`

f. (1 point) 19(f):

Solution:
`Int`

g. (1 point) 19(g):

Solution:
`T-Lam`

h. (1 point) 19(h):

Solution:
`T-App`

i. (1 point) 19(i):

Solution:
`Int -> Int`

j. (1 point) 19(j):

Solution:
`Int`

k. (1 point) 19(k):

Solution:
`T-Let`

Lambda Calculus Reference

```
-- Church numerals
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))

-- Booleans
let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let AND   = \b1 b2 -> ITE b1 b2 FALSE

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

-- Lists
let NIL   = \x -> TRUE
let CONS  = PAIR
let HEAD  = FST
let TAIL  = SND
let ISNIL = \lst -> lst (\h t -> FALSE)

-- Arithmetic
let SUC   = \n f x -> f (n f x)
let ADD   = \n m -> n SUC m

-- The definitions of DECR, SUB, ISZ, and EQL are elided
-- but you can still use them:
let DECR  = \n ->   -- (decrement n by one)
let SUB   = \n m -> -- (subtract m from n)
let ISZ   = \n ->   -- (return TRUE if n == 0 and FALSE otherwise)
let EQL   = \n m -> -- (return TRUE if n == m and FALSE otherwise)

-- Note: Since ZERO is the smallest Church numeral,
-- calls to DECR and SUB bottom out at ZERO.
-- For example, DECR ZERO evaluates to ZERO,
-- and SUB TWO THREE evaluates to ZERO.

-- The Y combinator
let Y = \step -> (\x -> step (x x)) (\x -> step (x x))
```

Haskell Reference

- ```
  map :: (a -> b) -> [a] -> [b]
  map f []     = []
  map f (x:xs) = f x : map f xs
  ```
- ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f b []     = b
  foldr f b (x:xs) = f x (foldr f b xs)
  ```
- `(+) :: Num a => a -> a -> a`

  Returns the sum of its two arguments, e.g.,

  ```
  > 3 + 4
  7
  ```
- `max :: Ord a => a -> a -> a`

  Returns the maximum of its two arguments, e.g.,

  ```
  > max 3 4
  4
  ```
- `(&&) :: Bool -> Bool -> Bool`

  The logical 'and' operation.

  ```
  > True && False
  False
  > True && True
  True
  ```
- `(++) :: [a] -> [a] -> [a]`

  Append two lists, e.g.,

  ```
  > [1,2,3] ++ [4,5]
  [1,2,3,4,5]
  > "apple" ++ "orange"
  "appleorange"
  ```
- `nub :: [a] -> [a]`

  Remove duplicate elements from a list, e.g.,

  ```
  > nub [1,2,3,4,3,2,1,2,4,3,5]
  [1,2,3,4,5]
  ```

- `(\\) :: [a] -> [a] -> [a]`

  Compute the difference of two lists. In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus `((xs ++ ys) \\ xs) == ys`, e.g.,

```
> ["a","b","c"] \\ ["a"]
["b","c"]
> ["a","b","c","a"] \\ ["a","c"]
["b","a"]
```

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

  Compare arguments for equality, e.g.,

```
> False == True
False
> "apple" == "apple"
True
```