# CSE114A, Spring 2024: Midterm Exam

Instructor: Lindsey Kuper

May 9, 2024

Student name:	
CruzID:	@ucsc.edu

This exam has 11 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 completely fill in the circle for the correct choice.
- You have 95 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.

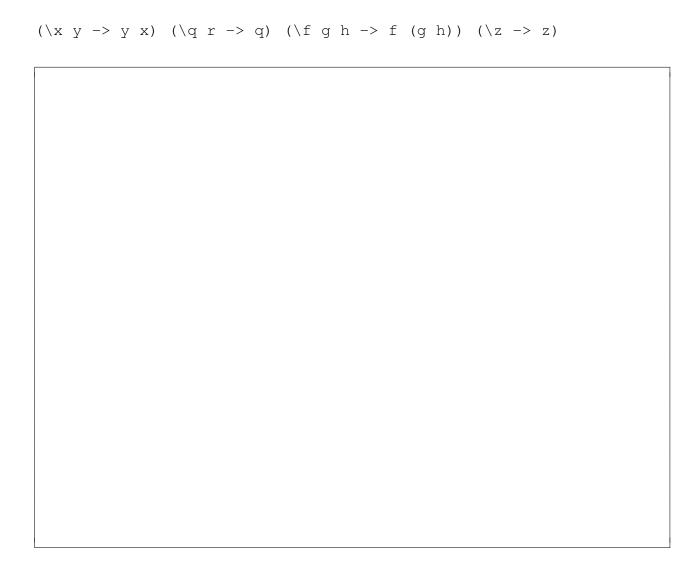
Good luck!

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

## Part 1: Lambda Calculus

1. (6 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  $\beta$ -reduction steps (and only  $\beta$ -reduction steps – you shouldn't need anything else). Start each line with =b>, as if you were using Elsa, and do just one  $\beta$ -reduction step per line.

Note: There may be multiple correct ways to reduce the expression. A correct solution is any solution that Elsa will accept as correct.



2. (5 points) Here is the BOX function:

let UNBOX =

```
let BOX = \x -> (\ignored -> x)
```

BOX takes an argument x, and returns a *box* containing x. A *box* is similar to a *pair*, except that it contains only one element.

Define a lambda calculus function UNBOX that takes a box as its argument and returns the contents of the box. For example, in Elsa:

```
UNBOX (BOX ZERO) = > ZERO

UNBOX (BOX TRUE) = > TRUE

UNBOX (BOX TWO) = > TWO

UNBOX (BOX (UNBOX (BOX TWO))) = > TWO
```

Hint: Think about how the FST and SND functions access the contents of a pair. (FST and SND are defined in the lambda calculus reference at the end of the exam.) Can you do something similar to access the contents of a box?

3.	Define a lambda calculus function NEST that takes a Church numeral n as an argument.	If n is
	ZERO, NEST returns ZERO. Otherwise, NEST n returns n nested boxes, with the innermo	ost box
	containing ZERO. For example, in Elsa:	
	NECH REDO "> REDO	

```
NEST ZERO =~> ZERO
NEST ONE =*> BOX ZERO
NEST TWO =*> BOX (BOX ZERO)
NEST THREE =*> BOX (BOX (BOX ZERO))
```

(Note: We are using = \* > instead of =  $^{\sim}$  > in the example calls to NEST above because the expressions can be further reduced. For example, BOX ZERO can be further reduced to  $\b$  -> ZERO, and BOX (BOX ZERO) can be further reduced to  $\b$  ->  $\b$  -> ZERO.)

You may assume that NEST is only called with non-negative integers represented as Church numerals. You may use any of the helper functions defined on the provided lambda calculus reference at the end of the exam. You must use recursion for full credit.

let NEST1 = \rec -> \n -> ITE(part 3(a)) (part 3(b)) (part 3(c))
let NEST =(part 3(d))
a. (5 points) 3(a):
b. (5 points) 3(b):
c. (5 points) 3(c):
d. (5 points) 3(d):

### Part 2: Haskell

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

4. (4 points) What is the **type** of the following Haskell expression?

5. (4 points) What is the **type** of the following Haskell expression?

6. (4 points) What is the **type** of the following Haskell expression?

7. (4 points) What is the **type** of the following Haskell expression?

man (\	(x -> 5)			expression	evaluate to	•	
map (	(x -> 5)		3, 4, 5]				
(5 points	s) What does	s the follow	ing Haskell	expression	evaluate to	?	
foldr	(++) <b>"</b> ge	ngar" [ˈ	"eevee",	"diglet	t <b>"</b> ]		

# **Part 3: Abstract Syntax Trees and Environments**

For all the questions in this section, we will use the following Expr data type. It defines the grammar of abstract syntax trees for SmolHaskell, a language that is a very (*very*) small subset of Haskell. SmolHaskell has numbers, Booleans, variables, addition expressions, subtraction expressions, and if-expressions, represented with the Expr *constructors* ENum, EBool, EVar, EPlus, EMinus, and EIf, respectively.

For example, we can represent the SmolHaskell program

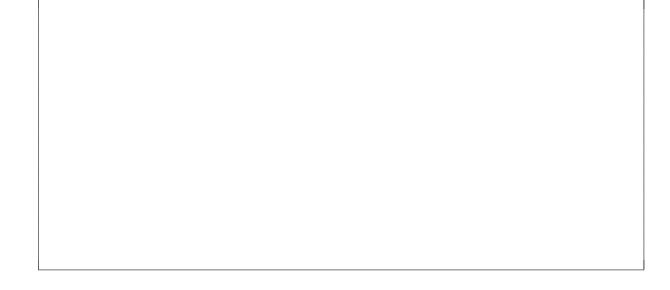
The *size* of an Expr is the number of Expr constructors that it has. For instance:

```
ENum 5 is size 1,
```

```
EPlus (ENum 3) (ENum 4) is size 3, and EIf (EBool False) (EPlus (ENum 5) (EVar "x")) (ENum 4) is size 6.
```

10. a. (12 points) Define a Haskell function size that takes an Expr and returns its size as an Int. You can use (+), but no other library functions. The type signature of size is provided for you below; fill in the rest of the definition.

```
size :: Expr -> Int
```



```
sizeAll :: [Expr] -> Int
sizeAll [] = 0
sizeAll (x:xs) = size x + sizeAll xs
```

Define a *tail-recursive* Haskell function <code>sizeAllTR</code> that has the same behavior as <code>sizeAll</code>. You may define a helper function or use a <code>where</code> clause. You can use (+), but no other library functions. (It's OK to use <code>size</code>, of course.) The type signature of <code>sizeAllTR</code> is provided for you below; fill in the rest of the definition.

```
sizeAllTR :: [Expr] -> Int
```



c. (7 points) Define a Haskell function sizeAllFoldr using foldr that has the same behavior as sizeAll. You can use foldr and (+), but no other library functions. (It's OK to use size, of course.) The type signature of sizeAllFoldr is provided for you below; fill in the rest of the definition.

Hint: The first argument to foldr should be of type Expr -> Int -> Int.

```
sizeAllFoldr :: [Expr] -> Int
```



11. (15 points) Finally, let's write an interpreter for Exprs!

Since SmolHaskell supports both numbers and Booleans, our interpreter should be able to return values of either type. For example, the SmolHaskell program

```
if True then False else True
evaluates to False, but the SmolHaskell program
if True then 3 else (4 + 5)
```

evaluates to 3. So, we will define a type for *values* that will encompass both numbers and Booleans:

```
data Value = VNum Int | VBool Bool
```

And now we can write an interpreter that returns a Value.

We also need to deal with Exprs that contain *variables*, so our interpreter will need to be an *environment-passing* interpreter, as seen on Assignment 3. We will represent an environment as a list of pairs of Strings and Values, using the following type alias:

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

So the type signature of our interpreter will be:

```
eval :: Expr -> Env -> Value
```

The environment associates variables with values. If we try to evaluate an expression containing a variable that does not have a binding in the provided environment, our interpreter should raise a run-time exception. The code that does this is in the provided helper functions, so if you use the helper functions, you won't need to implement the code that raises the exception yourself.

If we try to run a program that uses a number where it should use a Boolean, like if 3 then 5 else 7, or a program that uses a Boolean where it should use a number, like 3 + False, our interpreter should raise an run-time exception. Again, the code that does this is in the provided helper functions, so if you use the helper functions, you won't need to implement the code that raises the exception yourself.

Here are some example calls to eval:

```
> eval (EBool True) []
VBool True
> eval (EIf (EBool True) (EBool False) (EBool True)) []
VBool False
> eval (EIf (EBool False) (ENum 3) (EVar "x")) [("x", VNum 5)]
VNum 5
> eval (EIf (EBool True) (EVar "y") (ENum 3)) [("x", VNum 5)]
*** Exception: Unbound variable!
> eval (EIf (ENum 3) (ENum 5) (ENum 7)) []
*** Exception: Type error
> eval (EPlus (ENum 3) (EBool False)) []
*** Exception: Type error
```

We are now ready to define our interpreter. The type signature of eval and the first line of its definition is given for you below; fill in the rest of the definition.

Hint: Use (+), (-), and the helper functions at the bottom of the page, and you can do this in five lines of code.

```
eval :: Expr -> Env -> Value eval (ENum n) env = VNum n
```

#### 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 = \xy -> y
let ITE = \b x y \rightarrow b x y
-- Pairs
let PAIR = \xy \rightarrow (\b \rightarrow ITE b x y)
let FST = p \rightarrow p TRUE
let SND = p \rightarrow p FALSE
-- Boxes
let BOX = \x -> (\ignored -> x)
-- Arithmetic
let SUC = \n f x -> f (n f x)
let ADD = \n m -> n SUC m
-- The definitions of DECR, SUB, and ISZ 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)
-- 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 = \text{step} \rightarrow ((x \rightarrow \text{step} (x x))) ((x \rightarrow \text{step} (x x)))
```

#### **Haskell Reference**

```
• map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

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

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

Append two lists, e.g.,

• (==) :: Eq a => a -> a -> Bool

Compare arguments for equality, e.g.,

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