CSE4510/CSE5400 Take Home Final Fall 2016

1	2	3	4	5	6	7	8	9	10	total

name: Zubin Kadva / 902772316

Please work independently, but you make advantage of any written or Internet resources. Please give credit loosely for substantive help. Answer to each question on a separate page of paper.

1. (10 points). Put in all the parentheses in the following lambda expressions:

   (a) $(\lambda x.xa\lambda y.xy$

   (b) $(\lambda x.xz)\lambda y.w\lambda w.wyzx$

   (c) $\lambda x.xy\lambda x.yx$

2. (10 points). Apply beta reduction to the following lambda expression as much as possible. Show each step.

   (a) $(\lambda z.z)(\lambda y.yy)(\lambda x.xa)$

   (b) $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$

   (c) $(\lambda x.\lambda y.xyy)(\lambda y.y)y$

   (d) $(\lambda x.xx)(\lambda y.yx)z$

   (e) $(\lambda x.(\lambda y.(xy))y)z$

Answers

1.

   (a) $(\lambda x.xa)(\lambda y.xy)$

   (b) $(\lambda x.xz)(\lambda y.w)(\lambda w.wyzx)$

   (c) $(\lambda x.xy)(\lambda x.yx)$

2.

(a)	(b)	(c)	(d)	(e)
= $(\lambda y.yy)\ (\lambda x.xa)$ = $(\lambda x.xa)\ (\lambda x.xa)$ = $(\lambda x.xa)$ a = a a	= $(\lambda z.z)\ (\lambda w.ww)\ (\lambda x.xy)$ = $(\lambda w.ww)\ (\lambda x.xy)$ = $(\lambda x.xy)\ (\lambda x.xy)$ = $(\lambda x.xy)\ y$ = $y\ y$	= $(\lambda x.\lambda y.xyy)\ (\lambda w.w)\ z$ = $(\lambda x.\lambda y.xyy)\ z$ = $\lambda y.zyy$	= $(\lambda x.xx)\ zx$ = $z\ x\ z\ x$	= $(\lambda x.xy)\ z$ = $z\ y$

3. (10 points). Explain how a recursive function can be understood without resorting to machines or implementations.

Answer

A recursive function is used to describe a function that uses recursion. A recursive process is the one in which an object is expressed as another object of the same type. These objects require an initial value. Using some constraints, these objects can be constructed using recurrence relations.

Eg. To find sum of a list of numbers recursively:

sum [1, 2, 3, 4, 5]
= 1 + sum [2, 3, 4, 5]
= 1 + 2 + sum [3, 4, 5]
= 1 + 2 + 3 + sum [4, 5]
= 1 + 2 + 3 + 4 + sum [5]
= 1 + 2 + 3 + 4 + 5
= 3 + 3 + 4 + 5
= 6 + 4 + 5
= 10 + 5
= 15

(wikipedia.com, wolframalpha.com)

4. (20 points).

These questions concern eager evaluation.

(a) Define in Haskell a value of any type and call it b. One should have `b ::a`.

(b) Considering b just above, and k defined as follows

```
k a b = a
```

comment on the following Haskell boolean expressions:

```
b == 3
'a' == b
True && b
b && True
False && b
if True then 3 else b
if False then b else 'a'
if b then 17 else 17
k b 3
k 'a' b
k b
```

(c) Is there anything like b or k in the Haskell standard prelude?

(d) Is it true that for all Haskell expressions E_1 and E_2

```
False && E_1 == E_1
k E_1 E_2 == E_2
```

(e) Suppose (contrary to fact) that the functions k and (&&) were strict. Do the equations

```
False && E_1 == E_1
k E_1 E_2 == E_2
```

hold for E_1 and E_2

Answers

(a) Defining b as type Int. `b = 10`

(b)

b == 3 => False, since value of b is 10

'a' == b => Error, cannot compare a Char with Int

True && b => Error, since && operation requires a Boolean and b is Int

b && True => Error, since b is of type Int

False && b => Error, since b is of type Int

if True then 3 else b => 3

`if False then b else 'a'` =>Error, since b return Int and 'a' is of type Char i.e. type mismatch

`if b then 17 else 17` => Error, since 'if' requires a Boolean whereas b is an Int

k b 3 => 10, since function 'k' returns b whose value is 10

k 'a' b => 'a' since function 'k' returns its first argument viz. 'a'

k b => Error, since function 'k' requires 2 arguments and only 1 is passed

(c) Function 'k' is similar to the function '`const`'

```
> :t const
const :: a -> b -> a
```

(d)

`False && E_1 == E_1` => True, if E_1 is of type boolean

k E_1 E_2 == E_2 => True, if E_1 and E_2 are of same type

(e)

`False && E_1 == E_1` => False

k E_1 E_2 == E_2 => False

5. (10 points). Suppose

```
ones = [1..]
```

What is the value of `take 5 ones`? What is the value of `last ones`?

Define `ones` recursively without use the `..` notation.

A stream is like an infinite list.

```
data Stream a = Nil | Cons a (() -> Stream a)
```

Build an infinite stream of ones.

Answer

The value of `take 5 ones` is: [1, 2, 3, 4, 5]
Reason: Haskell uses lazy evaluation. Hence it computes the list of first 5 elements from an infinite list

The value of `last ones` is: <goes into an infinite loop>
Reason: Due to the list being infinite, Haskell's lazy evaluation evaluates 1, then 2, then 3 and so on. Thus resulting in a continuous loop.

Recursively, `ones` in defined as:

```
ones = 1 : map (+1) ones
```

Infinite stream of `ones`:

```
ones = Cons 1 (() -> Stream 1)
```

6. (20 points). Read Meijer's "Curse of the Exclude Middle".

   (a) What is big difference between polymorphism in Haskell and generics in Java?

   (b) What are non-proper morphisms?

   (c) What are the new "elephants" in the room?

   (d) Define the word *insouciant*.

   (e) Explain what the acronym DSL means.

   (f) Explain how to break to circumvent Haskell's type system.

   (g) Explain the title of the paper.

   (h) Summarize the main thesis of the article.

Answer

(a) Haskell's definition of the monad class is parameterized by a type constructor rather than a type. Generics allow parameterization with types eg. `List<T>` but not allow parameterization over the container type `M/` eg. `M<T>` and then instantiate `M` with `List`.

(b) A function which injects a pure value into a computation implies non proper mechanisms. These operations are unique to that effect / outcome eg. The monad `(STM a)` contains non proper morphisms for allocating, reading, writing of transactional variables.

(c) The new elephants in the room include concurrency and parallelism.

(d) The word *insouciant* means – unconcern, carefree or nonchalant *(dictionary.com)*

(e) DSL is an acronym for Domain Specific Language, i.e. a language specific to a domain. Functional languages are DSLs since they are just executional, denotational semantics.

(f) Consider the function `unsafePerformIO :: IO a -> a` that tells the compiler to forget about the consequences of evaluating its argument. Thus allowing any type to be cast to any other type and hence breaks the Haskell type system.

(g) The curse of the excluded middle is an attempt to create an intermediate language that effectively is in the "middle" of functional and imperative languages. It states – preventing effects by making them fully explicit in the type system or allow explicit effects to be suppressed.

(h) This article contrasts functional and imperative programming languages by explaining the effects of a mutating state with examples from lazy evaluation and that languages become safe when removing these effects.

7. (10 points). Imagine a robotic pencil that accepts four diff t commands: forward, backward, turn left and turn right. When the car turns left or right, it always turns exactly 90 degrees.

Suppose we model the commands

```
data Command
  = Forward Int
  | Backward Int
  | TurnLeft
  | TurnRight
```

The integer argument denotes the distance in that direction.

Implement a function

```
destination :: [Command] -> (Int,Int)
```

that, given a list of commands computes the position after following these commands. The original position of the car is (x,y)=(0,0), and it is facing upwards in the sense that going forwards will increase its y position.

Example:

```
*Main> destination [Forward 20, Backward 10, TurnRight, Forward 100]
(100,10)

*Main> destination [Forward 20, Backward 5, TurnLeft, Forward 100]
(-100,15)
```

Turn in your program with the submit server using the tag robot.

Answer

Submit server:
file: Main.hs
tag: robot

8. (20 points). Define a datatype representing propositions. Include "true" and "false" and the connectives: conjunction, disjunction, and negation. Make your datatype an instance of the class Arbitrary include order to use the QuickCheck package. (The package must be installed with Cabal.)

   Write a function `norm` to convert any proposition to one in negation normal form. Negation normal form has negation applied only to propositional variables and to no other propositional.

   Test your program using QuickCheck. Include in your program one or more tests like `test = quickCheck ...`

   Turn in your program with the submit server using the tag `norm`.

   Answer

   Submit server:
   file: Main.hs
   tag: norm

9. (15 points). Using the following data type:

```
data Term a = Var a | App (Term a, Term a) | Abs (Term (Bind a))
data Bind a = Zero | Succ a
```

represent each of the lambda expressions in problem 2.

In this representation bound variables introduced by lambda abstractions are represented by natural numbers. An occurrence of a number n in an expression represent the bound variable introduced by the n-th nested lambda abstraction.

So $\lambda x.\lambda y.x(yy)$ would be

```
Abs (Abs (App (Var Zero, App (Var (Succ Zero), Var (Succ Zero)))))
```

Free variable can be represented by their String name.

Answers

(a)

```
App (
    App (Abs (Var Zero), Abs (App (Var (Succ Zero), Var (Succ Zero)))),
    Abs (App (Var (Succ (Succ Zero)), Var (Succ "a")))
)
```

(b)

```
App (
    App (Abs (Var Zero), Abs (App (Var Zero, Var Zero))),
    Abs (App (Var Zero, Var (Succ "y")))
)
```

(c)

```
App (
    App (
        Abs (Abs (App (Var Zero, App (Var (Succ Zero), Var (Succ Zero))))),
        Abs (Var (Succ Zero))
    ),
    Var "y"
)
```

(d)

```
App (
    App (Abs (App (Var Zero, Var Zero)), Abs (App (Var Zero, Var (Succ "x")))),
    Var "z"
)
```

(e)

```
App (Abs (App (
        Abs (App (Var Zero, Var (Succ Zero))),
        Var (Succ Zero)
    )),
    Var "z"
)
```

10. (20 points). Consider the following definition for the types of a language. (The language has ML-like syntax.)

VAR $A \vdash V : T$ when $A(V) = T$

CON $A \vdash C : T$ when $A(C) = T$

APP $$\frac{A \vdash E : T' \to T \qquad A \vdash E' : T'}{A \vdash E\ E' : T}$$

TUP $$\frac{A \vdash E_1 : T_1 \qquad A \vdash E_2 : T_2 \ \ldots \ A \vdash E_n : T_n}{A \vdash (E_1, E_2, \ldots, E_n) : T_1 \times T_2 \times \ldots \times T_n}$$

COND $$\frac{A \vdash E_1 : \text{bool} \qquad A \vdash E_2 : T \qquad A \vdash E_3 : T}{A \vdash \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 : T}$$

LET $$\frac{A \vdash E' : T' \qquad A + [V : T'] \vdash E : T}{A \vdash \textbf{let val } V = E' \textbf{ in } E \textbf{ end} : T}$$

ABS $$\frac{A + [V : T'] \vdash E : T}{A \vdash \textbf{fn } V \Rightarrow E : T' \to T}$$

The types in the language are int,char, and function types. There are no type variables.

(a) Given A = [3 : int, a : char], prove $A \vdash$ let val f = fn$x$ => x in (f3, f3) end : int × int

(b) Prove that for all A and τ it is the *not* the case that $A \vdash$ let val f = fn$x$ => x in (f3, $f$$a$)end : τ

Answers

Next page

(a)

$$\dfrac{\dfrac{\dfrac{A = 3 : int}{A \vdash x : int}\,\text{VAR}}{A \vdash fn\ x => x : int}\,\text{TUP} \qquad \dfrac{\dfrac{A = a : char}{A \vdash (f3, f3) : char}\,\text{CON}}{A + [f : char] \vdash (f3, f3) : int \times int}\,\text{TUP}}{let\ val\ f = fn\ x => x\ in\ (f3, f3)\ end : int \times int}\,\text{LET}\ \text{LET}$$

(b)

Suppose $A \vdash let\ val\ f = fn\ x => x\ in\ (f3, fa) end : \tau$

Applying the LET rule, we get

$$A \vdash fn\ x : t \qquad\qquad A + [f : \tau] \vdash (f3, fa) : \tau$$

This is not possible because a function *fn x* cannot be assumed to be τ since we do not know what τ might contain.

Moreover, assuming f is of type τ, we cannot conclude that *(f3, fa)* is also of type τ.

Thus, the hypotheses does not hold for all values of A and τ.