Chris Fenton (fenwil28)
CNC - PLD
Winter Final

1. Lists in the lambda calculus. Here is the file list.hs. It includes the lambda terms we've been studying (slightly simplified: no constants or delta rules, and I provided some print functions for ease of checking the normal forms of some terms.) Using this I defined a set of lambda terms to implement lists. You can run this file and study the terms. After you have done so, complete the following :

(a) construct a term that represents the empty list.

```
empty' = Abs "f" (Abs "x" (V "z")) --same as false
emptyList = (App (App pair empty') empty')

Abs "z'" (App (App (V "z'") (Abs "f" (Abs "x" (V "z")))) (Abs "f" (Abs "x" (V "z"))))
```

(b) construct a term that represents a list with 3 elements.

```
mylist3 = Abs "z" (App (App (V "z") (V "a")) (Abs "z" (App (App (V "z") (V "b")) (V
"c"))))

Abs "z" (App (App (V "z") (V "a")) (Abs "z" (App (App (V "z") (V "b")) (V "c"))))
```

(c) get the head of your 3 element list, reducing it to its normal form. (Use the lambda term headl).

```
myhead3 = reduceAll $ App headl mylist3
```

(d) get the tail of your 3 element list, reducing it to its normal form. (Use the lambda term taill).

```
mytail3 = reduceAll $ App taill mylist3

Abs "z" (App (App (V "z") (V "b")) (V "c"))
```

(e) show that headl(consl a l) == a (for any term a and list l).

```
RHS:
= a
= V "a"  #rewrite in lambda form

LHS:
headl(consl a l) =
App headl (consl a l) =  #rewrite outer application in lambda form
App headl (Abs "z" (App (App (V "z") (V "a")) (V "l "))) =  #replace consl w/ def
App (Abs "p" (App (V "p") (Abs "x" (Abs "y" (V "x"))))) (Abs "z" (App (App (V "z") (V
"a")) (V "l ")))  #replace headl w/ def
V "a" = #after reducing
-- LHS = RHS
```

(f) show that taill(consl a l) == l (for any term a and list l).

```
-- RHS:
-- = l
-- = V "l"  #rewrite in lambda form

-- LHS:
-- taill(consl a l) =
-- App taill (consl a l) =  #rewrite outer application in lambda form
-- App taill (Abs "z" (App (App (V "z") (V "a")) (V "l "))) =  #replace consl w/ def
-- App (Abs "p" (App (V "p") (Abs "x" (Abs "y" (V "y"))))) (Abs "z" (App (App (V "z")
(V "a")) (V "l "))) =  #replace tail w/ def
-- V "l"  #after reduction
-- LHS = RHS
```

2. Ycombinator. The file list.hs also contains a number of definitions for simple recursive functions using the Ycombinator: add', map', and len. Complete the following:

(a) define the term for add 2 3

```
myfive = reduceAll (App (App (App y add') (App succl one)) (App succl (App succl one)))
```

(b) show that add 2 3 == 5 by reducing it to normal form.

```
Abs "s" (Abs "z" (App (V "s") (App (V "s") (App (V "s") (App (V "s") (App (V "s") (V
"z")))))))
```

(c) implement multiply using the same model as add (add using successor; you can use add to implement multiply).

```
grt' = (Abs "x" (Abs "z" (App isZero (App (V "x") (App predcn (V "z") )) )))
mult' = (Abs "f" (Abs "m" (Abs "n" (App (App (App if_cn (App (App grt' (V "n")) zero )
) (App (App (V "f") (App (App (App y add') (V "m") ) (V "n"))  ) (App predcn (V "n"))))
( V "m" )))))
```

Note: mult isn't quite right, but I spent too much time on this as it is. However, I'm pretty proud of getting a greater than function working and getting the expression as far as I did.

Better version fixes tries to apply the correct algorithm, but is even more broken:

```
mult'' = (Abs "f" (Abs "m" (Abs "n" (App (App (App if_cn (App (App grt' (V "n")) zero )
) (App (App (V "f") ( rc  ) ) (App predcn (V "n")) ))  ( V "m" ))))) where
   rc = (App (App add' (V "m") ) m' )
   m' = (App (App (App y mult'') (V "m") ) (App predcn (V "n")) )
```

(d) show that mul 2 3 == 6 by reducing it to normal form.

```
(((\f -> (\x -> (f (x x)) \x -> (f (x x))) \f -> \m -> \n -> (((\p -> \e1 -> \e2 -> ((p
e1) e2) ((\x -> \z -> (\n -> ((n \t -> \t -> \f -> f) \t -> \f -> t) (x (\n -> \s -> \z
-> (((n \g -> \h -> (h (g s))) \u -> z) \u -> u) z))) n) \s -> \z -> z)) ((f (((\f ->
(\x -> (f (x x)) \x -> (f (x x))) \f -> \m -> \n -> (((\p -> \e1 -> \e2 -> ((p e1) e2)
(\n -> ((n \t -> \t -> \f -> f) \t -> \f -> t) n)) m) ((f (\n -> \s -> \z -> (s ((n s)
z)) m)) (\n -> \s -> \z -> (((n \g -> \h -> (h (g s))) \u -> z) \u -> u) n)))) m) n))
(\n -> \s -> \z -> (((n \g -> \h -> (h (g s))) \u -> z) \u -> u) n))) m)) (\n -> \s ->
\z -> (s ((n s) z)) (\n -> \s -> \z -> (s ((n s) z)) \s -> \z -> z))) (\n -> \s -> \z
-> (s ((n s) z)) (\n -> \s -> \z -> (s ((n s) z)) (\n -> \s -> \z -> (s ((n s) z)) \s
-> \z -> z))))

** some magic happens **

\s -> \z -> (s (s (s (s (s (s (s (s z))))))))
```

3. Ycombinator – sort of ! The definitions of map' and succl make use of the Ycombinator because we do not have recursive definitions in the lambda calculus.

(a) Try to type the Ycombinator, and show where the problem is.

```
\f -> ( \x -> (f (x x)) \x -> (f (x x)) )  // \f:r.exp:r -> r'
\f:r -> (exp):r'
exp = ( \x -> (f (x x)) \x -> (f (x x)) )
exp = App e1 e2
e1:: \x:r'' -> (expr):r'''
expr = App e1 e2
expr:: f:r ->r' (exprr):r
exprr = App e1 e1  // here's the problem
exprr:: x:r->r' x:r // we need to apply x to itself
```

We need a special recursive type to be able to apply x to itself.

(b) Pretend that we can give a recursive definition directly (as we do in Haskell), so that we could have the term:

```
map' = λ g . λ l . if ( isnill l ) then l
       else Cons (g ( headl l )) (map' g (taill l ))
```

Now give the type constraints for map' succl applied to a list of Church numerals. You can assume that a Church numeral has the type Nat and that lists have the type [α] for some type α.

```
map' = λ g . λ l . if ( isnill l ) then l
       else Cons (g ( headl l )) (map' g (taill l ))

map' : t1 -> t2 // all functions have this type
g    : t1  // first argument in function, so gets t1
l    : t2
if   : t3 // let's give the body of the function a name

Useful stuff:
succl :: Nat -> Nat -> Nat
consl :: α -> α -> [α]
isnill :: [α] -> r -> r' or [α] -> Bool if we had a Bool type defined
if :: Bool -> α -> α -> α
headl :: [α] -> α
taill :: [α] -> [α]

Facts:

if :: Bool -> a -> a -> a // if takes a bool, then expr, else expr, and returns an a
then : l = t2
else : Cons (g ( headl l )) (map' g (taill l )) // in haskell we would already have a
type rule that constrains the then and else branches to have the same type, so we would
know that the else branch needed to be t2

Let's break down the else branch

Cons :: a -> a -> [a]
else : [a]

Since we g = succl and l = [Nat] we can further constrain our type variables

g    : Nat -> Nat -> Nat // from succl
l    : [Nat]
if   : Bool -> [Nat] -> [Nat] -> [Nat]

So we can constrain map'

Map' g l :: (Nat -> Nat -> Nat) -> [Nat] -> [Nat] // if must return a [Nat]
```

(c) Using unification and the constraints from the preceding step, derive the type of the term for map' succl applied to a list of Church numerals.

I think I rolled this into the previous question.

4. Clite meaning functions For the following clite program.

```
int main ( ) {
  int x ;
  int y ;
  x = 0 ;
  y = 3 ;
  while ( y > 0 ) {
    x = x + y ;
    y = y - 1 ;
  }
}
```

(a) Show the initial state

```
Environment = {<x,int,0>,<y,int,1>}
Memory      = {<0,undef>,<1,undef>}
```

(b) Show the meaning function for each iteration of the while loop.

WhileStatement semantics:

```
WhileStatement -> while (Expression) Statement

N State Expression Statement
```

Iterations	State	Expression	Statement
0	{<x,0>,<y,3>}	y > 0	Block: Assignment: x = x + y Assignment: y = y - 1
1	{<x,3>,<y,2>}	true	Block: Assignment: x = 3 + 2 Assignment: y = 2 - 1
2	{<x,5>,<y,1>}	true	Block: Assignment: x = 5 + 1 Assignment: y = 1 - 1
3	{<x,6>,<y,0>}	false	

The meaning is the state after the while loop executes.

5. More Typing Show the types generated by

```
λ f . λ x . ( f f ) x
λ f . λ x . f ( f x )
```

Determine which is well-typed (if any) and which isn't (if any) by demonstrating that the type constraints are (or are not) satisfiable.

```
λ f . λ x . ( f f ) x
```

I can already tell this isn't going to be well typed by self application in (f f). This is the same problem as the y combinator.

```
λ f . λ x . f ( f x )

Type variables:
f              = a
x              = b
f(f x)         = c -> d
f x            = e -> f
\f.(\x.f(f x)) = g -> h
\x.f(f x)      = i -> j

f x            = c \\app rule
f              = b ->c \\app rule
f(f x)         = c \\app rule
\x.f(f x)      = b -> c \\abs rule
\f.(\x.f(f x)) = (b -> c) -> b -> c \\abs rule
```

This looks well typed. It also looks a lot like the constant two and f x = f . x in haskell.