Box-and-arrow Diagrams

- Draw box-and-arrow diagrams for each of the following statements. What needs to be copied, and what can be referenced with a pointer?

  (define a ‘((squid octopus) jelly sandwich))
  (define b (car a))
  (cons b a)
  (cons a b)
  (append b a)
  (cons ‘(I will have the) b)
  (cadar b)
  (cons a ‘b)

Tail Recursion

- Here is a function that sums all of the even elements in a list

```
(define (sum-evens x)
        (cond [(null? x) 0]
              [(= 0 (modulo (car x) 2)) (+ (car x) (sum-evens (cdr x)))]
              [else (sum-evens (cdr x))]))
```

Why is this not tail recursive? Rewrite it so that it is.

Currying

1. [Winter 2014] Define a Haskell function uncurry that takes a curried function with two arguments and returns a function that takes a pair instead. Thus...

```
uncurry (+)
```

should be a function that takes a pair of numbers and adds them, so that

```
uncurry (+) (3,4)
```

Should evaluate to

```
7
```

What is the type of uncurry?

Common Functions (fold, map, filter, etc)

1. Write a Racket macro “my-map” that works like the map function - i.e. it takes in a function and a list of any number of elements and returns a list of the result of applying the function to those lists.

For example...

```
(my-map func: (lambda (x) (+ x 2)) elts: (1 2 3))
```

should return

```
‘(3 4 5).
```

2. Consider the Haskell list:

```
x = [1,4..]
```

What is the result of the following functions? If they result in infinite computation, give the first five values returned if applicable; else write “infinite calculation”. If they result in an error, say so and give a (very brief) justification why:

```
zip x [2,8]
foldr (+) 0 x
foldl (:) [] [(head x)..]
zip x x
filter (\x -> odd x) x
```

Haskell Constructs (Monads, infinite ds, list comprehension, etc)

1. What are the first 5 elements of these infinite lists:

```
w = map (\(x,y) -> x && y) $ repeat (True,False)
x = 1.0 : map (\x -> x/2) x
y = 1 : 2 : y
z = [ (a - b)^2 | a <- [1, 2..], b <- [-1,-2,..]]
```

2. For each of the following, write the definition of the infinite list:

```
[‘a’, ‘b’, ‘c’, ‘d’,...]
[(1,1), (2,1), (3,1), (4,1), (5,1)...]
```

3. Desugar the following do function:

```
prgrm = do
        putStrLn “Give me something: “
        x <- readLn
        if x < 5 then return x else prgrm
```

What’s the type of this function?

4. The haskell Prelude function fmap allows you to apply functions to elements bound to Monad types. So if readln binds IO 2 to x, fmap (+ 2) x returns IO 4. The type for fmap is:
fmap :: Functor f => (a -> b) -> f a -> f b

a. Write a Haskell IO Monad as a do function that, when run, prompts the user to give it a string, which it then prints in reversed order (use fmap and the Haskell function reverse), asks the user for another string, then concatenates the second string onto the reverse of the first string, printing out the result. The particulars of the messages printed out are up to you. What’s the type of this function?

b. Desugar the function in (a)

Racket Constructs (Macros, Structs, delay, Mutability, etc):

1. Define a struct point3d that has 3 fields x, y and z. Then make two functions, one that tells the distance between two points, the other flips the point to the other side of the origin.

For example:

```
dist (1, 0, 0) (2, 0, 0)
flip (2, 3, 4)
```

would become:

```
1
(-2, -3, -4)
```

a. Do this without mutating the struct (i.e. return a new struct each time the second function is called).
b. Do this with mutation.

2. Write a Racket macro “my-map” that works like the map function - i.e. it takes in a function and a list of any number of elements and returns a list of the result of applying the function to those lists.

For example...

```
(my-map func: (lambda (x) (+ x 2)) elts: (1 2 3))
```

should return

```
‘(3 4 5).
```

3. What do the following result in? If they cause a type error, say so:

```
(let ([x 5]
      [y 3])
   (let ([x 7]
         [y x])
     (+ x y)))

(let ([x 5]
      [y 3])
   (let* ([x 3]
          [y x])
     (+ x y)))

(let* ([x (lambda (y) (if (< y 1) y (x (- y 1))))])
   (x 5))

(letrec ([x (lambda (y) (if (< y 1) y (x (- y 1))))])
   (x 5))
```

4. Consider the following racket expressions:

```
(define a 5)
(define b 3)
(define c (+ a b))

(define (sleepy)
        (+ a c))

(define (raskell b)
        (- (sleepy) b))

(define (weird d a c)
        (+ (raskell a) d (sleepy)))
```

What would be the result of (raskell 12)? (weird 1 2 3)?

Types

Consider the following Haskell functions:

```
-- Thruple: Like a Tuple, but stores exactly 3 elements
data Thruple a = Garbage | Elts a a a
  deriving (Show,Read,Ord,Eq)

dumb_thruple_fn Garbage = Garbage
dumb_thruple_fn (Elts x y z)
  | x <= y && x <= z = Elts x x x
  | y <= x && y <= z = Elts y y y
  | z <= x && z <= y = Elts z z z

dumber_thruple_fn [] = []
dumber_thruple_fn (Garbage : ts) = []
dumber_thruple_fn ((Elts x y z) : ts) = (Elts x 0 0) : dumber_thruple_fn ts
```

For each of the following, mark **N** if it cannot model a type for the functions, **Y** if it can model a type but is not the most general type, and **G** if it models the most general type. The most general type may not appear in the list of each function:

a. dumb_thruple_fn :: Thruple a -> Thruple a
b. dumb_thruple_fn :: Thruple a -> Thruple b
c. dumb_thruple_fn :: Eq a => Thruple a -> Thruple a
d. dumb_thruple_fn :: Ord a => Thruple a -> Thruple a
e. dumb_thruple_fn :: (Ord a, Ord b) => Thruple a -> Thruple b
f. dumb_thruple_fn :: Thruple Int -> Thruple Int
g. dumber_thruple_fn :: [Thruple a] -> [Thruple a]
h. dumber_thruple_fn :: Fractional a => [Thruple a] -> [Thruple a]
i. dumber_thruple_fn :: Ord b => [Thruple a] -> [Thruple b]
j. dumber_thruple_fn :: (Ord a,Ord b) => [Thruple a] -> [Thruple b]
k. dumber_thruple_fn :: [Thruple Float] -> [Thruple Float]

Octopus

1. [Winter 2014] Consider the following OCTOPUS program

```
(let ((i 100)
      (f (lambda (i) (+ i 1))))
    (f (+ i 10)))
```

For the following questions, write out the environments as lists of name/value pairs, in the form used by the OCTOPUS interpreter. To simplify the task a little, you can just include a binding for + as the global environment, and omit the other functions and constants.

Hints: the global environment (simplified) is:

```
[ ("+", OctoPrimitive "+") ]).
```

The three parameters to OctoClosure are the list of the lambda's arguments (as strings), an environment, and an expression that is the body of the lambda.

(a) What is the environment bound in the closure for the lambda?

(b) What is the environment that OCTOPUS uses when evaluating the body of the function f when it is called in the above expression?

(c) What is the environment that OCTOPUS uses when evaluating the actual parameter to the call to f?