Box-and-arrow Diagrams

1. 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)
```

Solution: For the purposes of time (so we could get these out to you ASAP), since box-and-arrow diagrams are a time-consuming pain to get into a word document, we’ve chosen not to write these out for you, but if you find any TA they can verify your work/talk you through them if you have any questions.

Tail Recursion

1. 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.

Solution:

```
(define (sum-evens x)
  (acc-sum-evens x 0))

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

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?

Solution:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a,b) = f a b
```

Alternatively, you can have:

```
uncurry f = (\(x,y) -> f x y)
```

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).
```

Solution:

```
(define-syntax my-map
  (syntax-rules (func: elts:)
    ([my-map func: f elts:] '())
    ([my-map func: f elts: e1 e2 ...]
     (cons (f e1) (my-map func: f elts: e2 ...)))))
```

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:

Function:	**Solution**
zip x [2,8]	: [(1,2),(4,8)]
foldr (+) 0 x	: infinite calculation
foldl (:) [] [(head x)..]	: error!!!! Looks like [] : 1 : 2… - no base case []
zip x x	: infinite; take 5 of this results in [(1,1),(4,4),(7,7),(10,10),(13,13)]
filter (\x -> odd x) x	: infinite; take 5 of this results in [1,7,13,19,25]

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)
```
Solution: [False,False,False,False,False]

```
x = 1.0 : map (\x -> x/2) x
```
Solution: [1.0,0.5,0.25,0.125,6.25e-2]

```
y = 1 : 2 : y
```
Solution: [1,2,1,2,1]

```
z = [ (a - b)^2 | a <- [1, 2..], b <- [-1,-2..]]
```
Solution: [4,9,16,25,36]

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

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

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?

Solution:
The type of this program is IO Integer. Here’s a desugared version:

```
desugar_prgrm = putStrLn “Give me something: “ >> readLn >>=
        (\x -> if x < 5 then return x else prgrm)
```

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. The particulars of the messages printed out are up to you. What's the type of this function?

Solution 1:

```
silly_do :: IO ()
silly_do = do
  putStrLn "Give me a string: "
  x <- fmap reverse getLine
  putStrLn $ x ++ " now give me another string: "
  y <- getLine
  putStrLn $ x ++ y
```

Solution 2:

```
silly_do :: IO ()
silly_do = do
  putStrLn "Give me a string: "
  x <- getLine
  let y = reverse x
  putStrLn $ y ++ " now give me another string: "
  z <- getLine
  putStrLn $ y ++ z
```

b. Desugar part a.

Solution1:

```
desugar_silly =
            putStrLn "Give me a string: "
            >> fmap reverse getLine
            >>= (\x -> putStrLn (x ++ " now give me another string: ")
                        >> getLine
                        >>= (\y -> putStrLn $ x ++ y))

desugar_silly =
    putStrLn "Give me a string: "
    >> getLine
    >>= (\x -> putStrLn (reverse x ++ " now give me another string: ")
    >> getLine
    >>= (\z -> putStrLn $ reverse x ++ z))
```

Racket Constructs (Macros, Structs, delay, 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: the flipped version of

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

is

```
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

Solution:

```
(struct point3d (x y z) #:transparent #:mutable)

(define (dist3d p1 p2)
  (let ([xdist (- (point3d-x p1) (point3d-x p2))]
        [ydist (- (point3d-y p1) (point3d-y p2))]
        [zdist (- (point3d-z p1) (point3d-z p2))])
    (sqrt (+ (* xdist xdist) (* ydist ydist) (* zdist zdist)))))

(define (flip p)
  (point3d (- 0 (point3d-x p)) (- 0 (point3d-y p)) (- 0 (point3d-z p))))
```

Part b makes the struct mutable (just left that was in part a anyway) and changes flip:

```
(define (flip p)
  (set-point3d-x! p (- 0 (point3d-x p)))
  (set-point3d-y! p (- 0 (point3d-y p)))
  (set-point3d-z! p (- 0 (point3d-z p))))
```

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).
```

Solution:

```
(define-syntax my-map
  (syntax-rules (func: elts:)
    ([my-map func: f elts: ()] '())
    ([my-map func: f elts: (e1 e2 ...)] (cons (f e1) (my-map func: f elts: (e2 ...))))))
```

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)))
```

Solution: 12

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

Solution: 6

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

Solution ERROR

```
(letrec ([x (lambda (y) (if (< y 1) y (x (- y 1))))])
   (x 5))
```
Solution: 0

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)?
Solution: (raskell 12) would return 1. (weird 1 2 3) would return 25.

What if racket used dynamic scoping instead of static?
Solution: Let's walk through this:
(raskell 12): In raskell's environment, b = 12; this means that, when raskell calls sleepy, sleepy's environment is going to have b = 12 as well. Then sleepy adds a and c. a and c haven't changed, either, since c evaluated (+ a b) when it was defined. So, c is still 8, sleepy still returns 13, and then 13 - 12 is 1, so raskell 12 still returns 1.
(weird 1 2 3): In weird's environment, d = 1, a = 2, c = 3. When weird calls raskell, raskell sees these bindings in the environment, as well as that b = a = 2. So its call to (- (sleepy) b) represents (- (sleepy) 2), and sleepy sees all these bindings as well. sleepy computes (+ a c), which due to raskell's environment is (+ 3 2) = 5, so raskell has (- 5 2) = 3, which is returned as the result of (raskell a). (sleepy) called from weird's environment again has those values a = 2 and c = 3, which it adopts from weird, so it returns 5. So, the sum is 3 + 1 + 5 = 9.

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

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: [(OctoSymbol "+", 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?

Solution: [(OctoSymbol "+", OctoPrimitive "+")]

This is just the global environment. We evaluate the expressions for the values to be bound to each variable in the let in the enclosing environment for the let, which is the global environment in this case. f is bound to the result of evaluating (lambda ...). The lambda captures its environment of definition, which is thus the global environment.

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

Solution: [(OctoSymbol "i", OctoInt 110), (OctoSymbol "+", OctoPrimitive "+")]

When we evaluate a function, we extend its environment of definition, namely the ("+", OctoPrimitive "+") part of the above answer, with new variables for the formal parameters, bound to the actual parameters. Here, f has just one formal parameter, namely i. So the new environment has i as its first pair, with i bound to the value of the actual parameter. (See the next item for how this value is found.)

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

Solution: [(OctoSymbol "f", OctoClosure ["i"] [(OctoSymbol "+", OctoPrimitive "+")] (OctoList [OctoSymbol "+", OctoSymbol "i", OctoInt 1])), (OctoSymbol "i", OctoInt 100), (OctoSymbol"+", OctoPrimitive "+")]

This is the environment used to evaluate the body of the let, and consists of the global environment extended with bindings for the two variables in the let, namely i and f. So when we evaluate (+ i 10) we get 110.