CSE 341 - Programming Languages
Midterm - Winter 2015 - Answer Key

1. (4 points) What is the value of `mystery`? (If it's infinite give the first several elements.)

```
mystery = "squid" : mystery

["squid", "squid", "squid", "squid", "squid" .....]
```

2. (10 points) Define a function `nodups` in Racket that takes a sorted list of numbers and returns a list that is the same, except with duplicates removed from the list. For example, `(nodups '(1 1 1 1 3 4 4 4 5 5))` evaluates to `(1 3 4 5)`, and `(nodups '())` evaluates to `()`.

```
(define (nodups s)
  (cond ((null? s) s)
        ((null? (cdr s)) s)
        ((= (car s) (cadr s)) (nodups (cdr s)))
        (else (cons (car s) (nodups (cdr s))))))
```

3. (6 points) What is the result of evaluating the following Racket expressions?

```
(let ((a 1)
      (b 10)
      (c 20))
  (let* ((a 100)
         (b a))
    (list a b c)))

'(100 100 20)

(let ((n 10))
  (letrec ((f (lambda () (+ n 1)))
           (n 3))
    (f)))

4
```

4. (8 points) Consider the following Racket program.

```
(define y 10)

(define (clam)
   (+ y 1))

(define (crab y)
   (* y (clam)))
```

(a) What is the result of evaluating `(clam)`?
11

What is the result of evaluating `(crab 5)`?
55

(b) Suppose Racket used dynamic scoping. What would be the result of evaluating `(clam)`?
11
What would be the result of evaluating `(crab 5)`?
30

5. (6 points) Consider the `zip`, `zip3`, and `uncurry` functions from the Haskell Prelude. `zip` takes two lists and produces a single list, consisting of pairs of corresponding elements from each list. `zip3` does the same thing, but for three lists. `uncurry` takes an ordinary curried function with two arguments and turns it into a function that takes a single argument that is a pair. Finally, let's define a function `gt` that is an uncurried version of `>`. These are defined as follows:

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

zip3 [] _ _ = []
zip3 _ [] _ = []
zip3 _ _ [] = []
zip3 (x:xs) (y:ys) (z:zs) = (x,y,z) : zip3 xs ys zs

uncurry f (x,y) = f x y

gt x = uncurry (>) x
```

For example, `zip [1,2,3] [10,11,12]` evaluates to `[(1,10), (2,11), (3,12)]`, and `uncurry (+) (3,4)` evaluates to 7.

Circle each type declaration that is a correct type for `gt`. (Not necessarily the most general type, just a correct one.)

```
gt :: a -> b -> Bool  NOT CORRECT

gt :: Int a => a -> a -> Bool  NOT CORRECT

gt :: Ord a => (a,a) -> Bool CORRECT

gt :: Num a => (a,a) -> Bool  NOT CORRECT
(note that Num is not a subclass of Ord)

gt :: Integral a => (a,a) -> Bool  CORRECT
```

Which of the above types, if any, is the most general type for `gt`?
`Ord a => (a,a) -> Bool`

6. (5 points) Using the functions defined in Question 5, what is the type of each of the following Haskell expressions? If it has a type error, say that.

```
zip3
[a] -> [b] -> [c] -> [(a, b, c)]

uncurry
(a -> b -> c) -> (a, b) -> c

map gt zip
TYPE ERROR

map gt . zip
TYPE ERROR

map gt . (uncurry zip)
Ord t => ([t], [t]) -> [Bool]
```

7. (3 points) What is the value of each of these expressions? (They all are correctly typed.) If it is an infinite list, give at least the first 5 values in the list.

```
zip [1..] [20..25]
[(1,20),(2,21),(3,22),(4,23),(5,24),(6,25)]

map gt $ zip [1,2,3,4] [3,3,3,3]
[False,False,False,True]

zip3 [1..] [1..] [1..]
[(1,1,1),(2,2,2),(3,3,3),(4,4,4),(5,5,5) ...]
```

8. (7 points) Convert the following Haskell action into an equivalent one that doesn't use do.

```
  echo = do
      putStr "your input: "
      s <- getLine
      putStr "you typed "
      putStrLn s
```

```
echo = putStr "your input: " >> getLine >>= \s -> putStr "you typed " >>
       putStrLn s
```

9. (6 points) Consider the following OCTOPUS program.

```
(let ((n 100))
  (letrec
      ((f (lambda (m) (+ m n))))
    (f (+ n 5))))
```

(Note: this is corrected from the printed version, by adding the extra parens, as was written on the whiteboard.)

To simplify the answers a little, suppose that the global environment only contains bindings for +, −, and `equal?`. (Omit the other functions and constants.) So if the question were "What are the names in the global environment," the answer would be +, −, and `equal?`.

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

`n, f, +, -, equal?`

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

`m, n, f, +, -, equal?`

10. (10 points) Write a case for the OCTOPUS `eval` function to handle `or`. You can use a helper function if needed. Your code should have OCTOPUS handle `or` exactly as in Racket: it can take 0 or more arguments, and does short-circuit evaluation. Hints: `(or #f (+ 10 10) 3 #t)` evaluates to 20. Be sure you only evaluate `(+ 10 10)` one time. Here is the header for the new case:

```
eval (OctoList (OctoSymbol "or" : args)) env = .....
```

```
eval (OctoList (OctoSymbol "or" : args)) env = eval_or args env

eval_or [] env = OctoBool False
eval_or (x:xs) env =
    if exp1==OctoBool False
    then eval_or xs env
    else exp1
    where exp1 = eval x env
```

11. (10 points) True or false?

(a) In Racket, the expressions in the body of a `delay` will be evaluated zero times or one time, but never more than one time.

True

(b) In the Haskell expression `3+2.8`, the 3 is coerced from type `Int` to type `Float`.

False – unlike other languages, Haskell doesn't use coercion.

(c) In Racket, evaluating the expression `(cons 3 4)` results in an improper list.

True – it's the list `(3 . 4)`

(d) Suppose we have a Racket expression that uses `let*`, without any function definitions. With this restriction, if you replace the `let*` with `letrec`, the expression will always evaluate to the same thing.

False. Here's an example:

```
(define m 1)
(let* ((n m)
       (m 5))
  n)
```

This is correct for `let*` and evaluates to 1, but in Racket itself gives an error for `letrec` since `m` isn't defined at the time it is used. (Interestingly, in OCTOPUS it does work, due to the lazy construction of the environment, and gives a different answer, namely 5. But in any case it doesn't evaluate to the same thing.)

(e) In Racket, if `a` and `b` are both bound to symbols, `(equal? a b)` and `(eq? a b)` always evaluate to the same thing.

True