CSE 341 - Programming Languages
Midterm - Winter 2014

Your Name:

```
                                                    (for recording grades):
                                                    1. (max 12)

                                                    2. (max 5)
Total  (max 90):
                                                    3. (max 6)

                                                    4. (max 5)

                                                    5. (max 6)

                                                    6. (max 8)

                                                    7. (max 6)

                                                    8. (max 5)

                                                    9. (max 2)

                                                   10. (max 2)

                                                   11. (max 6)

                                                   12. (max 6)

                                                   13. (max 6)

                                                   14. (max 5)

                                                   15. (max 10)
```

You can bring a maximum of 2 (paper) pages of notes. No laptops, tablets, or smart phones.

1. (12 points) Consider the `zip` and `repeat` functions from the Haskell Prelude. We could define them as follows:

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

repeat x = x : repeat x
```

For example, `zip [1,2,3] [10,11,12]` evaluates to `[(1,10), (2,11), (3,12)]`.

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

```
zip :: [Int] -> [Int] -> [Int]

zip :: [Int] -> [Int] -> [(Int,Int)]

zip :: [a] -> [b] -> [(a,b)]

zip :: [a] -> [b] -> [(b,a)]

zip :: [a] -> [a] -> [(a,a)]

zip :: (Eq a) => [a] -> [a] -> [(a,a)]
```

Which of the above types, if any, is the most general type for `zip`?

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

```
repeat :: [Int] -> [Int]

repeat :: (Eq a) => a -> [a]

repeat :: [a] -> [[a]]

repeat :: a -> a
```

Which of the above types, if any, is the most general type for `repeat`?

2. (5 points) What is the value of each of the following Haskell expressions? If it has a type error, or if it gets a runtime error of some sort, say that. If it's an infinite list, give the first 5 values.

```
zip [1,2,3] [10,11]

zip "squid" "clams"

zip [1..] [10..]

zip (repeat True) (repeat "squid")

zip (1,2,3) (10,11,12)
```

3. (6 points) 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. (There is actually a built-in Haskell function called `uncurry` that does exactly this, but you should define it from scratch for this question.)

   What is the type of `uncurry`?

4. (5 points) What are the first 6 elements in the following Haskell infinite list?

```
mystery = 0 : (map (\x -> 2*x+1) mystery)
```

5. (6 points) Convert the following Haskell action into an equivalent one that doesn't use do.

```
echo = do
    j <- readLn
    k <- readLn
    putStrLn ("the sum is " ++ show (j+k))
```

6. (8 points) (Note: we are switching to Racket on this question!) Define a `zip` function in Racket, like the Haskell `zip` function from Question 1. For example,

```
(zip '(1 2) '(a b))
```

should return the list: `((1 a) (2 b))`.

Now define a curried version of `zip` in Racket.

7. (6 points) 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`?

8. (5 points) Suppose the following Racket code is evaluated. What is the output? If there is an error, be sure and give the output up to the point the error occurs.

```
(define d1 (delay (print "in d1") (newline) (/ 1 0)))
(define d2 (delay (print "in d2") (newline) (print d1) (newline)(/ 10 2)))
(print (force d2)) (newline)
(print (force d2)) (newline)
(print (force d2)) (newline)
```

9. (2 points) This question concerns fixed points and the `fix` function in Haskell:

```
fix f = f (fix f)
```

What is the value returned by evaluating each of these expressions in Haskell? If it gets into an infinite recursion, what *is* the fixed point of the function, if it has one? There might be zero, or many.

(a) `fix (const "squid")`

(b) `fix (^2)`

(c) `fix (+100)`

(d) `fix ("squid" : )`

10. (2 points) Here is a recursive version of the Fibonacci function in Haskell:

```
fib n = if n<2 then 1 else fib (n-1) + fib (n-2)
```

Write a nonrecursive version using `fix` as defined in Question 9.

11. (6 points)

(a) What does the following Racket program print out?

```
(define y 10)

(define (test1)
   (display y)
   (newline))

(define (test2 y)
   (display y)
   (newline)
   (test1))

(test1)
(test2 0)
```

(b) What would be printed if Racket used dynamic scoping?

12. (6 points) What is macro hygeine and why is it important?

13. (6 points) What is the difference between polymorphism and overloading? Give an example of each in Haskell.

14. (5 points) True or false?

    (a) In Haskell, if a type `Animal` is an instance of the `Show` typeclass, then there must be a `show` function that can be applied to instances of `Animal`.

    (b) The `show` function in Haskell works like a `show` method in Java — the system can check at compile time that there will be a `show` function available, but needs to wait until runtime to decide which one.

    (c) Any recursive function call in Racket will require stack space proportional to the number of recursive calls.

    (d) In Racket, if `(eq? expr1 expr2)` evaluates to `#f`, then `(equal? expr1 expr2)` will always evaluate to `#f` as well.

    (e) In Racket, if `(equal? expr1 expr2)` evaluates to `#f`, then `(eq? expr1 expr2)` will always evaluate to `#f` as well.

15. (10 points) Write a case for the OCTOPUS `eval` function to handle `and`. Your addition should make OCTOPUS handle `and` exactly as in Racket: it can take 0 or more arguments, and does short-circuit evaluation. Hints: `(and 1 #t 3)` evaluates to 3. Here is the header for the new case:

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