CSE 341 - Programming Languages
Midterm - Winter 2014 - Answer Key

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]  WRONG

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

zip :: [a] -> [b] -> [(a,b)] CORRECT AND MOST GENERAL

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

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

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

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] WRONG

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

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

repeat :: a -> a WRONG
```

Which of the above types, if any, is the most general type for `repeat`? **None of them!**

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]
[(1,10),(2,11)]
```

```
zip "squid" "clams"
[('s','c'),('q','l'),('u','a'),('i','m'),('d','s')]

zip [1..] [10..]
[(1,10),(2,11),(3,12),(4,13),(5,14), .....]

zip (repeat True) (repeat "squid")
[(True,"squid"),(True,"squid"),(True,"squid"),(True,"squid"),(True,"squid"), ...]

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

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.)

   ```
   uncurry f (a,b) = f a b
   ```

   What is the type of `uncurry`?

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

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

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

   ```
   [0,1,3,7,15,31]
   ```

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))

   desugared =
     readLn >>= \j -> readLn >>= \k -> 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))`.

   ```
   (define (zip x y)
     (cond ((null? x) '())
           ((null? y) '())
           (else (cons (list (car x) (car y)) (zip (cdr x) (cdr y))))))
   ```

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

```
(define (zip x)
  (lambda (y)
    (cond ((null? x) '())
          ((null? y) '())
          (else (cons (list (car x) (car y)) ((zip (cdr x)) (cdr y)))))))
```

This answer is also OK:

```
(define (curriedzip x) (lambda (y) (zip x y)))
```

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?

```
[ ("+", 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?

```
[ ("i", OctoInt 110), ("+", 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`?

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

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)
```

```
"in d2"
#<promise:d1>
5
5
5
```

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")` FIXED POINT IS `"squid"`

(b) `fix (^2)` INFINITE RECURSION - THERE ARE TWO REAL FIXED POINTS: 0 and 1

(c) `fix (+100)` INFINITE RECURSION - NO FIXED POINT

(d) `fix ("squid" : )` FIXED POINT IS `["squid", "squid", "squid", "squid", 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.

```
fib =  fix (\f -> \n -> if n<2 then 1 else f (n-1) + f (n-2))
```

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)
```

```
10
0
10
```

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

```
10
0
0
```

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

    A "hygienic" macro system gives fresh names to local variables in macros at each use of the macro, and binds free variables in macros where the macro is defined. Without hygiene, macro programmers need to use weird names for local variables and helper functions to avoid conflicts with names in the program that uses the macro. If there is a name collision, it will generally lead to incorrect or erroneous results.

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

    A polymorphic function is a single function that can be applied to arguments of different types. In Haskell, its type signature will include one or more type variables. An overloaded function is one where there are actually several different function definitions, and the system determines which to use based on the types of the arguments. In Haskell, `id` (the identity function) is a polymorphic function with type `a -> a`. The `show` function is overloaded — there are actually lots of different function definitions for it.

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`. **true**

    (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. **false** (Haskell also determines the function to use at compile time)

    (c) Any recursive function call in Racket will require stack space proportional to the number of recursive calls. **false** (a tail recursive function requires constant space)

(d) In Racket, if `(eq? expr1 expr2)` evaluates to `#f`, then `(equal? expr1 expr2)` will always evaluate to `#f` as well. **false** (For example, we might have two lists that are equal but not eq.)

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

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.

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

eval_and [] env = OctoSymbol "#t"
eval_and [x] env = eval x env
eval_and (x:xs) env = if (eval x env)==OctoSymbol "#f"
         then OctoSymbol "#f"
         else eval_and xs env
```