NAME

CSci 555: Functional Programming
Fall 2010, Examination #2

1. (30 points) Show the list yielded by each of the following Haskell list expressions. If the list is finite, display it using fully specified list bracket notation, e.g., expression `[1..5]` yields `[1,2,3,4,5]`. If the list is infinite, display the list using the ellipsis " $\cdots$ " appropriately. Assume that data type `Color` has been defined as follows:

```
data Color = Red | Orange | Yellow | Green | Blue | Violet | Grayscale Int
             deriving Show
```

   (a) `[4..9]`

   (b) `[9..4]`

   (c) `[9,6..1]`

   (d) `[ 2*i | i <- [1..10], odd i ]`

   (e) `take 5 [ n*n | n <- [2..], even n ]`

   (f) `[ j | i <- [1,-1,2,-2], j <- [1..i] ]`

   (g) `[ (x,y) | x <- [1..3], y <- [Blue,Red] ]`

   (h) `[ ys | (y:ys) <- ["Can", "you", "think", "recursively?"] ]`

   (i) `[ Grayscale x | x <- [1..]  ]`

   (j)
```
let  iterate f x  = x : iterate f (f x)
     sh []        = []
     sh (x:xs)    = xs
in   takeWhile    (/=[]) (iterate sh "ABCD")
```

2. (16 points) Remove the list comprehensions from the following expressions. That is, translate the list comprehensions into expressions using one or more of the functions `filter`, `map`, `foldr`, `++`, `fst`, `snd`, `zip`, etc.

   Functions `fst` and `snd` are prelude functions that extract the first and second components, respectively, from two-component tuples. Function `zip` returns a list of pairs of the corresponding elements of its two input lists.

   (a) `[x | x <- xs, p x]`

   (b) `[f (g x) | x <- xs]`

   (c) `[x | xs <- xss, x <- xs]`

   (d) `[ i | (i,x) <- zip [1..] xs, p x]`

3. (16 points) Consider the following definition for a factorial function `fact` in the Hugs version of Haskell, which uses an accumulating parameter. Note the pattern match and the use of the `strict` function.

```
fact :: Int -> Int -> Int
fact f 0 = f
fact f n = (strict fact (f*n)) (n-1)
```

Use string reduction as the model of computation.

(a) Briefly explain what is meant by normal order reduction? How are the redexes chosen for reduction? Does this correspond to eager evaluation or lazy evaluation?

(b) Show the normal order reduction of the expression `fact 1 3`.

(c) What is maximum space used by the above reduction?

(d) When will normal order graph reduction produce a result in fewer steps than normal order string reduction?

4. (10 points) The functional composition combinator is defined as follows:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

id :: a -> a
id x       = x
```

(a) Prove that functional composition is associative. That is, for all `f :: c -> d`, `g :: b -> c`, `h :: a -> b`, and `x :: a`, `((f . g) . h) x = (f . (g . h)) x`. (Hint: Ask yourself whether you need induction?)

(b) Also prove that `id` is the identity element for functional composition. That is, for any `f :: a -> b` and `x :: a`, prove `(id . f) x  =  f x  =  (f . id) x`.

5. (4 points) One of the problem-solving strategies we discussed is "solving a harder problem first". Briefly describe this strategy.

6. (3 points) Suppose we have the following Haskell definitions.

```
x = 1:y
y = map f x
f = (*2)
g = take 10 x
```

What would be displayed on the screen when `g` is evaluated?.

7. (25 points) An *S-expression* (i.e., symbolic expression as in the language Lisp) consists of a *number*, a *symbol*, or a *sequence* of S-expressions.

   If we use matched pairs of parentheses to denote sequences, then the S-expression `(3 (4 x) y)` consists of a sequence of three elements. The elements, in order, are the number `3`, the sequence `(4 x)`, and the symbol `y`. The sequence `(4 x)` itself consists of the number `4` and the symbol `x`. (Note: The notation in this paragraph is not intended to be Haskell.)

   An empty sequence of S-expressions is called a *nil*.

   Now consider how to represent these S-expressions in Haskell. Let an object of type `Sexpr` represent an S-expression:

```
data Sexpr = Num Int | Sym String | Seq [Sexpr]
             deriving Show
```

   The constructor `Num i` denotes a number with value `i`. The constructor `Sym s` denotes a symbol with value `s`. The constructor `Seq xs` denotes the sequence of S-expressions given in the Haskell list `xs`. If `xs` is `[]`, then the sequence is a nil. Thus the S-expression `(3 (4 x) y)` is represented in Haskell as `(Seq [Num 3, Seq [Num 4, Sym "x"], Sym "y"])`.

   SELECT FIVE of the following functions and show Haskell definitions and type signatures. You may use functions defined earlier in the list to define later ones.

   (a) Function `isNil` takes an S-expression and returns `True` if the `Sexpr` is a nil and `False` otherwise.

   (b) Function `cons` takes an S-expression `x` and a sequence `y` and returns the sequence in which `x` has been inserted in front of the elements of `y`. For example, `cons (Num 1) (Seq [Num 2])` returns `(Seq [Num 1, Num 2])`.

   (c) Function `car` takes a non-nil sequence and returns the first S-expression in the sequence. For example, `car (Seq [Num 1,Num 2])` returns `(Num 1)`.

   (d) Function `cdr` takes a non-nil sequence and returns the sequence remaining after removing the first element. For example, `cdr (Seq [Num 1, Num 2])` returns `(Seq [Num 2])`.

   (e) Function `equals` takes two S-expressions and returns `True` if they are exactly the same and returns `False` otherwise.

   (f) Function `append` takes an S-expression `x` and an S-expression `y` and returns the sequence in which the elements of `y` are appended after the elements of `x`. For example, `append (Seq [Num 1,Num 2]) (Seq [Num 3])` returns `(Seq [Num 1,Num 2,Num 3])`.

   (g) Function `rev` takes an S-expression (e.g., a sequence) and returns the S-expression with the elements in reverse order. For example, `rev (Seq [Num 1, Seq [Num 0, Num 3], Num 2])` returns `(Seq [Num 2, Seq [Num 0, Num 3], Num 1])`.