Solutions to Worksheet 5: Higher Order Functions and Lazy Evaluation

Due: Midnight, Monday, 4 April 2010.
Total marks: 59

Question 1: Typing Tutor (15 marks)

In this question, you were asked to deduce the type signature of a number of functions.
The solution is as follows:

```
-- (a) (1 mark)
boring :: Integer -> Integer -> Integer
boring x y = x + y

-- (b) (1 mark)
yawn :: Integer -> Integer -> Bool
yawn x y = x == y

-- (c) (2 marks)
stretch :: (t1 -> t2) -> t1 -> t2
stretch x y = x y

-- (d) (2 marks)
pull :: (t1 -> Integer) -> (t1 -> Integer) -> t1 -> Integer
pull a b c = (a c) + (b c)

-- (e) (3 marks)
twin :: (t1 -> t2 -> t3) -> (t -> t1) -> (t -> t2) -> t -> t3
twin u v w x = u (v x) (w x)

-- (f) (3 marks)
knit :: (t1 -> t1 -> t2) -> (t -> t1) -> t -> t -> t2
knit u v x y = u (v x) (v y)

-- (g) (3 marks)
-- given the following data definition. . .
data Temp = Cold | Cool | Warm | Hot
  deriving (Show)

-- . . . give the type signature for this function:
kelvin :: Integer -> Temp -> Integer -> Temp
kelvin a b c
  | a < 100 = Cold
  | otherwise = if (a > c) then Hot else b
```

This solution can be found in the file `tutor_sol.hs`. The solution assumes that arithmetic or comparison is done using `Integer` numbers (this was for simplicity, since there's no value in asking you to learn fine details about Haskell's type hierarchy).

Evaluation

You will receive full marks if your type signatures are correct, as shown.

Each type signature must show the types of each input parameter, and the output parameter.

No marks will be deducted if the type given is more general than the solution, as long as the solution is correct.

Marks will be deducted for errors, so partial credit will be given. A single mark will be deducted for each incorrect component. The minimum grade on each part is zero.

A missing type signature will receive a grade of zero.

Grading: To grade this question, check that the signatures are in a file along with the implementation, and that the signatures are uncommented. It should be possible to check the signatures without compiling the file, but compiling can be used as a check. If the signatures are commented out, then it's likely that the type is incorrect.

I expect pretty close to full grades on this quesiton, provided it was handed in.

Comments

I mentioned in class that Haskell can deduce the type signature for your functions. If you leave off an explicit type signature, and if your program compiles, you can ask Haskell for its inferred type. For example:

```
*Main> :l tutor.hs
[1 of 1] Compiling Main             ( tutor.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t boring
boring :: Num a => a -> a -> a
*Main> :t yawn
yawn :: Eq a => a -> a -> Bool
*Main> :t stretch
stretch :: (t1 -> t) -> t1 -> t
```

The `:type` command (abbreviated by `:t`) is a command for the interactive system only, ans is not part of the Haskell language. But it will report the type of the expression you give it. Notce that the type tht Haskell finds for the original questions are slightly more general than the solutions. This is due to the assumption that we assume Integers for arithmetic. Haskell's type signature will attempt to deduce the most general type, and the inference mechanism consists of facts about known functions, the hierarchy of types that are predefined, and a unification allgorithm much like Prolog's search engine.

Question 2: Getting into shape (27 marks)

This question consisted of some exercises involving *higher-order functions*, specifically `(.)`, `map`, `filter`, and `foldr`.

Consider the following datatype definition, similar to what we've seen in lecture, and the function `area`:

```
-- a pair for a 2D coordinate system
type Coord a = (a,a)

-- 4 kinds of shapes
data Shape a = Circle (Coord a) a          -- Circle center radius
             | Square (Coord a) a          -- Square corner side
             | Rectangle (Coord a) a a     -- Rectangle corner width height
             | Triangle (Coord a) a a      -- Triangle corner base height
  deriving (Show)


area :: Floating a => Shape a -> a

area (Circle _ r) = pi*r*r
area (Square _ s) = s*s
area (Rectangle _ w h) = w*h
area (Triangle _ b h) = b*h/2

test_shapes :: Floating a => [Shape a]
test_shapes = [Circle (1, 2) 3
              ,Square (4, 5) 6
              ,Rectangle (7, 8) 9 10
              ,Triangle (7, 5) 3 2
              ]
```

(You can find this program in the file `shapes.hs`, along with the type signatures for all the exercises that follow.)

1. (3 marks) Write a recursive function called `total_area` that takes a list of shapes, and computes the total area of all the shapes in the list. This function should not use any higher order functions at all.

   Solution:

   ```
   total_area :: Floating a => [Shape a] -> a
   total_area [] = 0
   total_area (s:ss) = (area s) + (total_area ss)
   ```

   Grading: Give full marks if the function is recursive, and calculates the correct value. A demonstration of the function is also necessary. Deduct a mark for missing demonstration, and for any error in the program.

2. (3 marks) Write a function called `total_area_2` using *higher-order functions* that takes a list of shapes, and computes the total area of all the shapes in the list.

   Solution:

```
total_area_2 :: Floating a => [Shape a] -> a
total_area_2 = foldr ((+).area) 0

-- an alternative definition
total_area_3 = (foldr (+) 0) . (map area)
```

Notice two key points: it's a function (by virtue of the type signature) but the argument is not explicitly named in the solution. Also note the fact that it makes one pass through the list, applying 2 functions in sequence: First `area` and then `(+)`.

Grading: For full marks, the function must use `foldr` and must also use `(.)`. Deduct a single mark if the function names its argument explicitly. An acceptable solution is

```
total_area_3 = (foldr (+) 0) . (map area)
```

since it uses `foldr` and `(.)`. The following solution:

```
total_area_4 list = foldr (+) 0 (map area list)
```

is operationally correct, but it names the argument (`list`), and so gets one mark deduction. It does not use `(.)`, either, but don't deduct a mark for it.

Solutions that use `sum` should receive at most 1 mark, since `sum` precludes the use of `foldr`.

3. (1 mark) Write a called `moveShape` that takes a point (`Coord a`) and a shape (`Shape a`), and moves the shape by amount given. Your function should have the following type:

   Solution:

```
moveShape :: Num a => Coord a -> Shape a -> Shape a
moveShape c (Circle d r)      = Circle    (addCoords c d) r
moveShape c (Square d r)      = Square    (addCoords c d) r
moveShape c (Rectangle d w h) = Rectangle (addCoords c d) w h
moveShape c (Triangle d t h)  = Triangle  (addCoords c d) t h

addCoords :: Num a => Coord a -> Coord a -> Coord a
addCoords (x,y) (u,v) = (x+u,y+v)
```

   This was not an interesting function, but it was needed for what follows, and so it's worth on mark. I also defined a function `addCoords` which will be useful later. It should ideally be located with the type definition for `Coord a`, but I leave it here because it's used here first.

   Grading: Give the mark for anything that adjusts the coordinate coorectly. There's no need to break it into 2 functions like I did. The function should have 4 patterns, though.

4. (3 marks) Write a function called `moveAll` that takes a list of shapes and moves them by `(10,10)`. Use higher-order functions, as in part 2 above, and your function `moveShape`.

   Solution:

```
moveAll :: Num a => [Shape a] -> [Shape a]
moveAll = map (moveShape (10,10))
```

Notice that the function does not name its argument, but it is a function. The implicit argument comes from the fact that the call to `map` leaves off the final argument as well. So the implicit argument for `moveAll` comes from `map`.

Define this function so that it does not explicitly name its arguments.

Grading: For full marks, the function must use `map` and `moveShape`. Give zero marks if it does not use both. Deduct a mark if the function definition explicitly names the argument. Deduct a mark if there is no demonstration of the function working.

5. (1 mark) Write a Boolean function called `isSquare` that returns `True` if given a shape that is a square, and `False` otherwise.

   Solution:

```
isSquare :: Shape a -> Bool
isSquare (Square _ _) = True
isSquare _ = False
```

Another boring function. It's possible to define a pattern for every shape, but in this given definition, the three non-squares are summarized by a single pattern.

Grading: Give the mark for anything that returns the correct answer, including functions with 4 patterns defined.

6. (3 marks) Using higher-order functions, define a function to return only the squares contained in a given list. This is fairly easy, so don't worry about looking for tricks here.

   Test your function on a small number of test lists.

   Define this function so that it does not explicitly name its arguments.

   Solution:

```
keepSquares :: [Shape a] -> [Shape a]
keepSquares = filter isSquare
```

This is a fairly simple use of filter. Again, notice the fact that the argument to `keepSquares` is implicit: it comes from `filter`.

Grading: For full marks, the function must use `filter` and `isSquare`, and it must not name the argument explicitly. Also, a demonstration must be given. It's possible to define this function using `foldr`, and don't deduct marks for that if it's done correctly. The code for `filter` in terms of `foldr` is in the notes.

7. (3 marks) Using higher-order functions, define a function to remove the squares contained in a given list. This is not as easy as the previous one.

   Hint: think of the process as keeping non-squares.

Test your function on a small number of test lists.

Define this function so that it does not explicitly name its argument.

Solution:

```
removeSquares :: [Shape a] -> [Shape a]
removeSquares = filter (not.isSquare)
```

The key here is the use of the compose operator `(.)`. Again, the argument is implicit.

Grading: For full marks, the definition must use `filter` and `(.)`, and must not name the argument explicitly. Also, there must be a demonstration of it working.

Deduct a mark if the program did not use `(.)` as in the following example

```
removeSquares_2 = filter notASquare
  where notASquare x = not (isSquare x)
```

This is not as good because it is twice as long as it needs to be.

8. (3 marks) Using higher-order functions, define a function to return all the squares and triangles contained in a given list. This is not as easy as the previous part.

   Hint: This one is more difficult to write all in one line. It is possible, but you may define helper functions. The point of this exercise is to see the difference between something you can do simply (as in previous parts) and when you have to do a bit more work.

   Test your function on a small number of test lists.

   Define this function so that it does not explicitly name its arguments. You may use helper functions (but you don't have to).

   Solution: The whole point of this exercise is to see that there is a kind of threshold in the sophistication of the use of functions like `(.)`. You have to recognize when using `(.)` saves you time and effort, and when it's simpler to build a function without using `(.)`. This is a case where it's possible, but extremely silly to use `(.)`.

```
keepSquaresAndTriangles :: [Shape a] -> [Shape a]
keepSquaresAndTriangles = filter special
  where special x = (isSquare x) || (isTriangle x)

isTriangle :: Shape a -> Bool
isTriangle (Triangle _ _ _) = True
isTriangle _ = False
```

Again, the argument to `keepSquaresAndTriangles` is implicit.

Grading: For full marks, the definition must use `filter`, and `isSquare` and something like `isTriangle`. There also has to be a demonstration. Deduct a mark if the demonstraion is missing, or if `filter` is not used. A solution like the one below using `twin` is acceptable, but not expected.

Comments The main call to `filter` looks a lot like `keepSquares`, but the test predicate, `special` looks for squares or triangles. There is no simple way to use `(.)` with `(||)` and `isSquare` to build a funciton equivalent to `special`. The problem is that a single shape has to be passed to 2 different functions, before getting passed to `(||)`.

On the other hand, observe the use of `twin` from Question 1:

```
keepSquaresAndTriangles_2 = filter (twin (||) isSquare isTriangle)
  where twin u v w x = u (v x) (w x)
```

Here I put `twin` in a local definition. I do not suggest that this is an ideal solution. If you think of `(.)` and `twin` as adapters, you might quickly realize that having an adapter for every purpose is clearly impossible, so keeping a toolbox full of them is a burden. Use the ones that help you, and do everything else as simply as possible.

9. (4 marks) This part is a warm up to the last part of this question.

   (a) (1 mark) First, write a function called `myAve1` to find the average of a list of `Float` numbers. Use `length` and `sum`. Note that in this version, you are (probably) traversing the list twice, which is not too bad, but can be improved.

   Don't worry about empty lists, or division by zero. It's not part of the learning objective here.

   Solution:

```
myAve1 :: [Float] -> Float
myAve1 list = (sum list) / (fromInteger (mylength list))
  where mylength = foldr ((+).(\x -> 1)) 0
```

   A boring function. The complication of using `fromInteger` was an unanticipated obstacle. Even I get used to implicit type conversion.

   Solution: Give the mark for anything that reasonably used `sum` and `length`. The type conversion problem made this question way more difficult than I had intended, so be generous with the mark.

   (b) (3 marks) Now write a function called `myAve2` that uses `foldr` (without using `length` or `sum`) that traverses the input list only once. To do this, you will need to keep track of both the sum and the count of the elements in the list. Use a pair of `Float` for this, as in the following example:

   Solution:

```
myAve2 :: [Float] -> Float
myAve2 l = a / b
  where (a,b) = foldr f (0,0) l
        f x (s,c) = (s+x,c+1)
```

   Grading For full marks, the definition must use `foldr` and a pair of values, as given in the hint. Because the hint was so strong, give zero marks for any solution tht did not use it. It is acceptable to define the combining function (`f` above) separately.

Comments There are a number of interesting points here. First is the use of a pattern to access data from a pair (the where-clause).

The second is the use of a pair in the call to `foldr`. This is a very important technique, sometimes called *tupling* (because a pair is a special case of a 2-tuple). It allows `foldr` to collect or accumulate more than a single piece of data.

The implication is that, with tupling, you only have to pass through a list once, to accumulate multiple values. It gives Haskell functions the same efficiency as loops, or tail recursion in Prolog. However, it takes some practice to get right, because it's very easy to forget which order functions like `f` (above) needs its arguments. But using `foldr` (along with some of its colleagues like `foldl`, `scanr`, etc) this way replaces loops and explicit recursion in many programs.

10. (3 marks) Using the ideas in the previous part, write a function called `borg` that takes a list of shapes as input, and creates a circle whose area is the sum of the areas of the shapes in the list, and whose position is, arbitrarily and nonsensically, the sum of the positions of the shapes. Your function should traverse the list exactly once.

    Solution:

```
borg :: Floating a => [Shape a] -> Shape a
borg ss = Circle coords (sqrt (a/pi))
  where (coords,a) = foldr f ((0,0),0) ss
        f s (c,r) = (addCoords (getCoord s) c, r + (area s))

getCoord :: Shape a -> Coord a
getCoord (Circle (x,y) _)       = (x,y)
getCoord (Square (x,y) _)       = (x,y)
getCoord (Rectangle (x,y) _ _) = (x,y)
getCoord (Triangle (x,y) _ _)  = (x,y)
```

    In this program, note the use of tupling: the call to `foldr` uses a tuple consisting of `((0,0),0)` i.e., a `Coord` and a single value. I could also have used a triple, e.g., `(0,0,0)`. I also defined a function `getCoord`, which should actually be defined somewhere closer to the definition of `Coord a`. Note the use of `addCoords` which I defined in Q2.3.

    Grading: Give full marks if the definition uses `foldr`, and if it uses tupling to collect multiple values in a single pass of the list. Deduct 2 marks if the definition scans more than one list one time. For example, it's not acceptable to split the list into two, and scan both. That's 3 scans. There should be a demonstration, but because I didn't ask explicitly, marks should not be deducted if the demonstration is not there.

Evaluation

You will receive no credit if you do not use higher order functions when required (as indicated above). Several parts ask for a definition that does not explicitly name its parameters, and one mark will be deducted if your definition does explicitly name the parameters for these questions.

Question 3: A rose, by any other name... (5 marks)

This question is intended to demonstrate the convenience of using list comprehensions, rather than recursive functions.

Write an function that builds the same list as the following list comprehension. You may not use comprehensions or higher order functions of any kind; you are allowed to use functions you have defined yourself recursively.

```
mylist n = [(x,y)| x <- [1 .. n], y <- [x .. n], (x + y) `mod` 3 == 0 ]
```

(You can find this listing in the file `comp.hs`.) Hand in any functions you may have written, and a demonstration that your function produces the required output.

Solution: Here are two possible approaches to the problem. Note that both solutions are much more work than the comprehension. That's the point, right?

```
-- Recursive function to compute the same output:
-- This version uses a single function to control the repetition
-- loop starts with a pair, as an accumulator
--       if the pair is valid, it's put into the list
--       if the pair is not valid, then one or both of the components
--       are changed
mylist2 :: Integer -> [(Integer,Integer)]
mylist2 n = loop (1,1)
  where loop (x,y)
          | x > n = []
          | y > n = loop (x+1,x+1)
          | (x + y) `mod` 3 == 0 = (x,y):loop (x,y+1)
          | otherwise = loop (x,y+1)


-- Recursive function to compute the same output:
-- This version has two explicit loops
--   loop1 controls the first componenet of the pair
--   loop2 controls the second componenet of the pair
--         and checks the validity of the pair
mylist3 :: Integer -> [(Integer,Integer)]
mylist3 n = loop1 1
  where loop1 x
          | x > n = []
          | otherwise = (loop2 x) ++ (loop1 (x+1))
          where loop2 y
                  | y > n = []
                  | (x + y) `mod` 3 == 0 = (x,y):loop2 (y+1)
                  | otherwise = loop2 (y+1)
```

Grading: Give full marks if the function is correct, and are accompanied by a correct type signature. A missing type signature will result in a deduction of 1 mark. If there is no demonstration of the function working correctly, deduct 1 mark.

The output does not need to be the same order as the given comprehension.

The name for this question...

Clearly, Shakespeare was wrong. Consider a list to be named by the program that generates it. A list named by a list comprehension is much plainer and more succinct than a list named by an equivalent collection of for-loops or recursive calls. Thus the name of the thing does make a difference.

Question 4: Infinitely Lazy Lists FTW (12 marks)

Consider the following function which is written using patterns and recursion:

```
sumFromZero :: Integer -> Integer
sumFromZero 0 = 0
sumFromZero x = x + (sumFromZero (x-1))
```

It's actually not a very interesting program. Now consider the following definition of an infinite list called sums:

```
sums :: [Integer]
sums = sumo 0 1
  where sumo s i = s : sumo (s+i) (i+1)
```

The list is defined in terms of a function, `sumo`, which has no base case. It takes 2 arguments, `s` the current sum, and `i` which is the current index. The value of `s` is put into the list, and the recursive call prepares the next value for the list by updating the sum, and incrementing the index.

The function `sumo` is very much like a tail recursive program, in that it uses accumulators (s and i). However, it's not tail recursion in the sense we discussed with Prolog, primarily because the function is creating an infinite list.

As you may remember from lecture, the infinite list is not created all at once, but evaluated one element at a time *as needed*. If you only need part of the list, only the part you ask for is created. For example:

```
*Main> take 10 sums
[0,1,3,6,10,15,21,28,36,45]
```

(the function `take` is built-in, and returns the first $n+1$ elements of a list). But be aware that if you ask for the whole list, you will start an infinite calculation:

```
*Main> sums
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91,
105, 120, 136, 153, 171, Interrupted
```

By typing `Ctrl-C`, you can interrupt a long or infinite calculation.

Haskell has a built-in function to get the nth element of a list; it's an infix function called `!!`, which starts counting at 0:

```
*Main> sums !! 100
5050
```

Complete each of the following exercises, based on the example of `sums`, above. Your solutions should use infinite (lazy) lists as above, or else you will receive no credit.

1. (3 marks) Using the example of `sums` above, define an infinite list consisting of the factorial numbers, e.g.,

```
*Main> take 10 facts
[1,1,2,6,24,120,720,5040,40320,362880]
*Main> facts !! 20
2432902008176640000
```

Hint: The first time you define an infinite list, it's a bit of a mind bender, because you've been told never to write recursion without a base case, and you've never created lists like this before. Follow the example carefully.

Solution:

```
facts :: [Integer]
facts = facto 1 1
  where facto p n = p : facto (p*n) (n+1)
```

Grading: For full marks, the definition needs a type signature, and should define a recursive function with no base case. Deduct a mark for if the type signature is missing, or if there is a base case. There must be 2 accumulators here.

Comment In this example, the recursive helper uses 2 accumulators. The first is the product, and the second is an index. Both are needed to compute the next product and index. Note also that the technique puts one of the accumulators on the list, and calculates updated accumulators to pass to the recursive call. It's very similar to tail recursion, except that it isn't.

2. (3 marks) Using the example of `sums` above, define an infinite list consisting of the Fibonacci numbers, e.g.,

```
*Main> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
*Main> fibs !! 100
354224848179261915075
```

Hint: You'll have to decide what initial values to send to your recursive function, and how to update them. It's not difficult, but approach it logically, rather than intuitively. For `fibs`, the sequence does not need an index. Write out the list, so you can see the patterns.

Solution:

```
fibs :: [Integer]
fibs = fibo 0 1
  where fibo f1 f2 = f1 : fibo f2 (f1+f2)
```

Grading: For full marks, the definition needs a type signature, and should define a recursive function with no base case. Deduct a mark for if the type signature is missing, or if there is a base case. There must be 2 accumulators here.

Comment Here the 2 accumulators represent Fibonaccis numbers. Specifically, the next one, and the one after it. There is no need to keep track of the index of the Fibonacci numbers (unlike factorial, which needed an index for future calculations). This solution is as efficient as any loop in any programming language, and it's the method of choice for calculating Fibonacci numbers: keep marching up until you get the one you want. Unlike a for loop, the Haskell program for generating Fobinacci numbers does not need to worry about which one to generate. The Haskell code only worries about how to calculate any Fibonacci number. Someone else (e.g., the use of `(!!)` at the command line) worries about which one. This is a nice way to separate concerns that you just don't get with a for loop..

3. (3 marks) Using the example of `sums` above, define an infinite list consisting of approximations to the square roots of any given number: e.g.

```
*Main> take 5 (sqrts 10)
[1.0,5.5,3.659090909090909,3.196005081874647,3.16245562280389]
*Main> (sqrts 10) !! 5
3.162277665175675
*Main> take 5 (sqrts 4)
[1.0,2.5,2.05,2.000609756097561,2.0000000929222947]
*Main> (sqrts 4) !! 5
2.000000000000002
```

Your definition should define an infinite list of approximations as generated through Newton's method:

$$\begin{aligned} y_0 &= 1 \\ y_i &= \frac{y_{i-1} + x/y_{i-1}}{2}, \quad i > 0 \end{aligned}$$

As for `fibs`, you do not need an index for `sqrts`.

Hint: This part requires you to write a function, whose value is an infinite list. The function's input should be x, the number whose square root you are approximating. The output list should be a sequence of approximations to the square root of x.

Solution:

```
sqrts :: Floating a => a -> [a]
sqrts x = sqrto 1
  where sqrto y = y : sqrto ((y + (x/y))/2.0)
```

Grading: For full marks, the definition needs a type signature, and should define a recursive function with no base case. Deduct a mark if the type signature is missing, or if there is a base case. Only one accumulator is needed here. Deduct a mark if more than one accumulator is used, as this was explicitly stated as a hint.

Comment A key to this solution is the use of scoping. The helper function does not need to have `x` passed into it, because it is local to `sqrts` and can see the value of `x`. The helper function also does not need an index, and in fact should only have one accumulator. We do not need the index because the index is not really part of the calculation, it's part of the description of the method. Using recursion like this allows us to write the method without using an explicit index.

4. (3 marks) Using the example of `sums` above, define an infinite list consisting of pseudo-random numbers, as generated by a linear congruent generator, e.g,

```
*Main> take 5 (randoms 123 2345 34567 456789)
[456789,16017,2117,20767,33295]
*Main> (randoms 123 2345 34567 456789) !! 1001
15487
```

A *linear congruential generator* (LCG) is a simple way to generate pseudo-random numbers. Given an initial value s, called the *seed*, we can define the sequence as follows:

$$\begin{aligned} r_0 &= s \\ r_i &= (r_{i-1} \times a + b) \bmod c, \quad i > 0 \end{aligned}$$

where a, b, c are given constants, which should be chosen carefully.[1]

Solution:

```
randoms :: Integer -> Integer -> Integer -> Integer -> [Integer]
randoms a b c s = rando s
  where rando x = x : rando (mod (x*a + b) c)
```

Grading: For full marks, the definition needs a type signature, and should define a recursive function with no base case. Deduct a mark if the type signature is missing, or if there is a base case. Only one accumulator is needed here. Deduct a mark if more than one accumulator is used, as this was explicitly stated as a hint.

Comment Like the previous example, the helper does not need to be passed the parameters of the method, i.e., `a b c s`. These values are visible to the helper because the helper is local. The helper function merely puts the current value of the accumulator into the list, and uses the formula to compute the next value of the accumulator.

If you've done the exercise properly, the last two examples should have been very easy.

Evaluation

Your definitions must be essentially similar to the above. Type signatures are trivial, but include them for good practice. No credit will be given if you do not define infinite lazy lists.

[1] LCG is not a really good way to generate random numbers, but it is simple, and makes for a good exercise.