Problem Set 1

1. Write a function `isFibonacci n` which takes an integer `n` and gives as output `True` or `False` depending on whether or not the supplied number is part of Fibonacci series. Assume:

```
Fib 0 = 0
Fib 1 = 1
Fib n = Fib (n-1) + Fib (n-2)
```

2. Consider the following function:

```
ack 0 n = n + 1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

   How will you describe

```
(a) ack 2 n, and
(b) ack 3 n
```

   as mathematical functions of `n`?

3. Write a function `where_smallest f a b` which takes a function `f` and the integer range `[a, b]` and determines the integer point in the given range where the function has the smallest value.

4. Given two numbers `a` and `b`, write a function `coeffs` which will return a pair of numbers `x` and `y` such that $ax + by = gcd(a, b)$.

5. Write a function `solution a b c` which returns the integer solutions `(x,y)` of the diophantine equation $ax + by = c$ if it has a solution, and `error ""` otherwise.

   HINT: The diophantine equation $ax + by = c$ has a solution for integer values of x and y, if $gcd(a, b)$ divides c.

6. Write a function `ak_mult x y` to multiply two numbers using the Al-Khwarizmi method. The Al-Khwarizmi method repeatedly halves the first argument and doubles the second argument, adding the second argument to the result, if the first is odd. Here is how we multiply 19 and 23:

```
    19 * 23
=   9  * 46 + 23
=   4  * 46 + 23 + 46
=   2  * 92 + 23 + 46
=   1  * 184 + 23 + 46
=   253
```

7. Define a function `mydiv x y`, $y \neq 0$, which will return a pair of numbers (q, r) such that $x = yq + r$ and $r < y$. This should also work by repeated halvings of x.

8. Given two integers `x` and `n` and an integer exponent `y`, write a function `modexp x y n` which will output: $x^y \ mod \ n$. This should also work by repeated halvings of y.

9. A Carmichael number is a non-prime number p such that $\forall a \in [1, p]$, whenever a is relatively prime to p, $a^{(p-1)} = 1 \ mod \ p$. Define a function `carmichael n` which will give the nth Carmichael number.

10. Suppose we wanted to write the following functions, explained through examples of how they may be called:

    (a) `f0 1 100`: Starting with 1, the sum of every 13th number in the interval 1 to 100.

    (b) `f1 2 100`: The sum of squares of all odd numbers between 2 and 100.

    (c) `f2 1 100`: The product of the factorials of multiples of 3 between 1 and 100.

    (d) `f3 10 200`: The sum of the squares of the prime numbers in the interval 10 to 200.

    (e) `f4 50`: The product of all positive integers less than 50 that are relatively prime to 50.

    Write a higher-order function called filtered-accumulate so that the functions described above are instances of filtered-accumulate. Write the functions f0, f1, f2, f3 and f4 in terms of filtered-accumulate. You can assume that the arguments to f0, f1, f2, f3 and f4 are positive.

11. We want to implement sets. Since the most important thing you can do with a set is to test membership of a given element, we want to represent sets as the datatype `Set(a -> Bool)`. The function `a -> Bool` associated with a set is called the *charateristic function of the set* in set-theoretic jargon. Under such a representation, the empty set is represented as `Set(\x  -> False)`. This is because the empty set returns false when tested for membership with any element. Similarly the complement of `Set f` would be `Set(\x -> not(f x))`.

    Under such a representation, write definitions for the following set theoraetic operations:

    - insert (inserts an element in the set)
    - member(tests for membership)
    - union
    - intersection
    - difference

    What is the advantage of such a representation over a list based representation?

12. Write a function `convert` which will convert a at-most-six-digits number into its verbal description. For example, `convert 308000` returns `"three hunderd and eight thousand"`, `convert 369027` returns `"three hunderd and sixty nine thousand and twenty seven"` and `convert 369401` will returns `"three hunderd and sixty nine thousand and four hundred and one"`.

13. We can represent any natural number using only `Zero` and a successor function, `Succ n`, where `n` is also in this form. This representation is also known as *Peano numbers*. Your task it to create your own data type in Haskell for Peano numbers and code the following:

    - A function which converts an integer to a Peano number and also write the dual of this function. For example:
      `peanoToInt 2 = Succ (Succ Zero)`, and `intToPeano (peanoToInt 2) = 2`
    - A mutually recursive version of even and odd function for Peano numbers.

- Make your data type an instance of `Num` class and implement the different operators defined in this class for Peano numbers.
  To start off, the data type should be defined somewhat like this:
  `data Peano = Zero | Succ Peano`