Function properties

Some properties of reverse

```
reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse (x:xs) = (reverse xs) ++ [x]
```

```
reverse [x] = [x]
```

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

```
reverse (reverse xs) = xs
```

As Haskell predicates

```
prop_RevUnit x =
   reverse [x] == [x]

prop_RevApp xs ys =
   reverse (xs ++ ys) == (reverse ys) ++ (reverse xs)

prop_RevRev xs =
   reverse (reverse xs) == xs
```

QuickCheck properties

squares_prop.hs

```
import Test.QuickCheck

square :: Integer -> Integer
square x = x * x

pyth :: Integer -> Integer -> Integer
pyth a b = square a + square b

prop_square :: Integer -> Bool
prop_square x =
   square x >= 0
```

```
prop_squares :: Integer > Integer –> Bool
prop_squares x y =
   square (x+y) == square x + square y + 2*x*y

prop_pyth :: Integer -> Integer -> Bool
prop_pyth x y =
   square (x+y) == pyth x y + 2*x*y
```

Running the program

```
dclark3:demo davidclark$ ghci squares_prop.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( squares_prop.hs, interpreted )
Ok, modules loaded: Main.
*Main> quickCheck prop_square
Loading package transformers-0.2.2.0 ... linking ... done.
Loading package mtl-2.0.1.0 ... linking ... done.
Loading package extensible-exceptions-0.1.1.2 ... linking ... done.
Loading package old-locale-1.0.0.2 ... linking ... done.
Loading package time-1.2.0.3 ... linking ... done.
Loading package random-1.0.0.3 ... linking ... done.
Loading package array-0.3.0.2 ... linking ... done.
Loading package containers-0.4.0.0 ... linking ... done.
Loading package pretty-1.0.1.2 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.4.0.1 ... linking ... done.
+++ OK, passed 100 tests.
*Main> quickCheck prop_squares
+++ OK, passed 100 tests.
*Main> quickCheck prop_pyth
+++ OK, passed 100 tests.
```

crossword helper

```
cwordFind :: Char -> Int -> Int -> [String] -> [String]
cwordFind letter pos len words =
    [wd|wd <- words, length wd == len, wd !! pos == letter]

cwordFindRec :: Char -> Int -> Int -> [String] -> [String]
cwordFindRec letter pos len [ ] = [ ]
cwordFindRec letter pos len (x:xs) =
    if (x !! pos == letter) && (length x == len) then
        x:(cwordFindRec letter pos len xs) else
        cwordFindRec letter pos len xs

cwordFindHO :: Char -> Int -> Int -> [String] -> [String]
cwordFindHO letter pos len words =
    filter p words
        where p x = (x !! pos == letter) && (length x == len)
```

```
prop_cwfCompRec :: Char -> Int -> Int -> Bool
prop_cwfCompRec letter pos len =
     cwordFind letter pos len == cwordFindRec letter pos len

prop_cwfRecHO :: Char -> Int -> Int -> Bool
prop_cwfRecHO letter pos len =
    cwordFindRec letter pos len == cwordFindHO letter pos len
```

Proving properties

For recursive functions we can use induction on the definition of the function to prove that the function has a property

If the data used by the function is recursive we can use induction on the structure of the data to show that the function has the property

Examples

```
- - pre condition x, y >= 0
- - post condition mult x y = x*y

mult :: Int -> Int -> Int
mult x 0 = 0
mult x y | y % 2 == 0 = 2 * mult x (y `div` 2)
         | otherwise  = x + mult x (y `div` 2)
```

```
- - pre condition n > 0
- - post condition sumTo n = sum (1 to n)

sumTo :: Int -> Int
sumTo 1 = 1
sumTo n = n + sumTo n - 1
```

Correctness for recursive functions

Recursive functions are shown correct by demonstrating two things:

1. A recursive variant
2. A proof by induction that the function establishes the post condition.

The recursive variant is a non negative expression: using the input parameters which gets "smaller" each time the function is called.

- pre condition: logical statement that holds on the data to which the function is applied
- post condition: logical condition that holds on the data after the function has been applied
- Given that the pre condition holds we show that the function establishes the post condition

The meaning of "smaller" will usually depend on the context.

Examples of variants:

for sumTo the variant is just x

for mult the variant is just y

The idea is that the variant gets smaller until the base case is reached and the call terminates.

Using induction

There are many forms of induction.
In particular there are two equivalent forms of mathematical induction

Simple Induction:

to show a property P holds on all natural numbers
first show the base case: P(0)
then show the inductive step:
that if you have a proof of P(n-1) you can prove P(n)

Note that there is no need for a base case!!

Course of values:

to show a property P holds on a set of natural numbers

show that if you know P(m)

for every m < N

then you can show P(N).

Simple induction. Show sumTo n = sum (1 up to n)

Basis: n = 1. sumTo(1) = 1 = sum (1 up to 1)

Induction: assume sumTo(n-1) = sum (1 to n-1) [IH]

sumTo(n) = n + sumTo(n-1)
= n + sum (1 to n-1)
= sum (1 to n)

So the function works for every integer bigger or equal to 1

- - mult pre: x,y>=0, post: mult x y =x*y

Course of values induction.

[IH] assume mult x m = x*m for every m < N, where N > 0

Two cases:

N % 2 = 0

two sub cases:

N = 0: mult x 0 = 0 = x*0

N > 0:

mult x N = 2 * mult x (N `div` 2) = 2 * (mult x N/2)

= 2* x * N/2 = x * N

$$N \,\%\, 2 = 1$$

$$\begin{aligned} \text{mult } x\ N &= x + 2 * (\text{mult } x\ \ (N \text{ `div` } 2)) \\ &= x + 2 * \text{mult } x\ (N\text{-}1)/2 \\ &= x + 2 * x * (N\text{-}1)/2 = x + x * (N\text{-}1) \\ &= x * N \end{aligned}$$

Note that the bottom element of the chain re-appeared as a special case here. In course of values induction this is how the base cases appear, rather than as a separate step.

```
- - pre: x > = 0, y > 0
- - post: remainder x y = x % y

remainder :: Int -> Int
remainder x y | x < y = x
              | otherwise remainder (x-y) y
```

Course of values induction proof

Induction Hypothesis: when n < N, remainder n y = n % y

case 1 N < y

remainder N y = N, by definition of remainder
= N % y, by definition of %

case 2 N >= y

remainder N y = remainder (N - y) y, by definition of remainder
= (N - y) % y, by I.H.
= N % y, property of %

Tail Recursion

Recursive functions can be distinguished as either

tail recursive, or
non tail recursive

What's the difference?

A tail recursive function has an inductive step that calls the the function directly on a "smaller" value (the tail). Non tail recursive functions do some computation with the recursive call

we have seen:

remainder	tail recursive
mult	non tail recursive
sumTo	non tail recursive

Tail recursive has a computational advantage. In effect tail recursion says "do the same computation again but with different arguments". Because of this advantage it is useful to be able to transform a non tail recursive procedure into a tail recursive one.

The usual way this is done is to define a new recursive procedure which has an extra argument.

Accumulators

An accumulator is a **parameter** of a recursive function that **accumulates the answer** as the function recurses towards the base case.

In general, the accumulator gets bigger as the variant gets smaller.

Let us look again at the function sumTo, calling it on parameter 5.

sumTo 5 = 5 + sumTo 4

4 + sumTo 3

3 + sumTo 2

2 + sumTo 1

1

The value
of each call
cannot be
calculated until
the recursion has fully unwound

sumTo(5) = 5 + sumTo 4

= 5 + (4 + sumTo 3)

= 5 + (4 + (3 + sumTo 2))

= 5 + (4 + (3 + (2 + sumTo 1)))

= 5 + (4 + (3 + (2 + 1)))

= 5 + (4 + (3 + 3))

= 5 + (4 + 6)

= 5 + 10

= 15

9 stack operations

```
- - pre: x >= 1, acc = 1
- - post: acc = sum (1 to x)

tailSumTo :: Int -> Int -> Int
tailSumTo 1 acc = acc
tailSumTo x acc = tailSumTo (x-1) (x + acc)
```

```
tailSumTo 5 1 = tailSumTo 4 6
              = tailSumTo 3 10
              = tailSumTo 2 13
              = tailSumTo 1 15
              = 15
```

No stack operations !!

Note importance of correctly initialising the accumulating parameter

examples: proofs by induction

```
sum :: [Integer] -> Integer
sum [ ] = 0
sum (x:xs) = x + sum xs

doubleAll :: [Integer] -> [Integer]
doubleAll [ ] = [ ]
doubleAll (x:xs) = 2*x : doubleAll xs
```

expect:
sum (doubleAll xs) = 2 * sum xs

```
- - QuickCheck version
prop_SumDoubleAll :: [Integer] -> Bool
prop_SumDoubleAll xs = sum (doubleAll xs) == 2 * sum xs
```

show base case holds:
sum (doubleAll [ ]) = 2 * sum [ ]

show inductive step holds:
assume sum (doubleAll xs) = 2 * sum xs Inductive Hypothesis (IH)
prove sum (doubleAll x:xs) = 2 * sum x:xs

(base case)

LHS:
sum (doubleAll [ ])
= sum [ ] by def doubleAll
= 0 by def sum

RHS:
2 * sum [ ]
= 2 * 0 by def sum
= 0 by arithmetic
= LHS

(induction case)

LHS:
sum (doubleAll x:xs)
= sum (2 * x : doubleAll xs) by def doubleAll
= 2*x + sum (doubleAll xs) by def sum

RHS:
2 * sum (x:xs)
= 2 * (x + sum xs) by def sum
= 2*x + 2* sum xs by arithmetic
= LHS by IH

```
length :: [a] -> Integer
length [ ]  = 0
length x:xs = 1 + length xs

(++) :: [a] -> [a] -> [a]
[ ] ++ zs = zs
(w:ws) ++ zs = w : (ws ++ zs)
```

length (xs ++ ys) = length xs + length ys

QuickCheck version:

```
prop_lengthPlusPlus :: [a] -> [a] -> Bool
prop_lengthPlusPlus =
    length ( xs ++ ys) == length xs + length ys
```

base case:
show length ([ ] ++ ys) = length [ ] + length ys

inductive case:
show length ((x:xs) ++ ys) = length (x:xs) + length ys

Inductive hypothesis (IH):
length (xs ++ ys) = length xs + length ys

Base case

LHS:
length ([ ] ++ ys)
= length ys by def ++

RHS:
length [ ] + length ys
= 0 + length ys by def length
= length ys by arithmetic
= LHS

Inductive case

LHS:
length ((x:xs) ++ ys)
= length x:(xs ++ys) by def ++
= 1 + length (xs ++ ys) by def length

RHS:
length (x:xs) + length ys
= 1 + length xs + length ys by def length
= LHS by IH

try these

sum (xs ++ ys) = sum xs + sum ys

xs ++ [ ] = xs

xs ++ (ys ++ zs) = (xs ++ ys) ++ zs

sum (reverse xs) = sum xs

Recall:

```
reverse [ ] = [ ]
reverse (x:xs) = reverse xs ++ [x]
```

Write two recursive procedures
Called fac and tailFac that implement the factorial function

e.g. fac 3 = 6

e.g. tailFac 4 = 24

show that they are correct