Specifying Pre- and Postconditions of Functions

Jan van Eijck

Specification and Testing, Week 2, 2014

Literate Programming, Again

```
module Week2

where

import Data.List
import Data.Char
import System.Random

infix 1 ==>

(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q

forall = flip all
```

Test Properties

Let `a` be some type.

Then `a -> Bool` is the type of `a` properties.

An `a` property is a function for classifying `a` objects.

Properties can be used for testing.

Pre- and Postconditions of Functions

Let f be a function of type $A \to A$.

A precondition for f is a property of the input.

$\text{pre} : A \to \{T, F\}$.

A postcondition for f is a property of the output:

$\text{post} : A \to \{T, F\}$.

Haskell type for pre- and postconditions: `a -> Bool`.

This is the type of properties of objects of type `a`.

Hoare Statements or Hoare Triples

$$\{p\}\ f\ \{q\}.$$

Intended meaning: if the input x of f satisfies p, then the output $f(x)$ of f satisfies q.

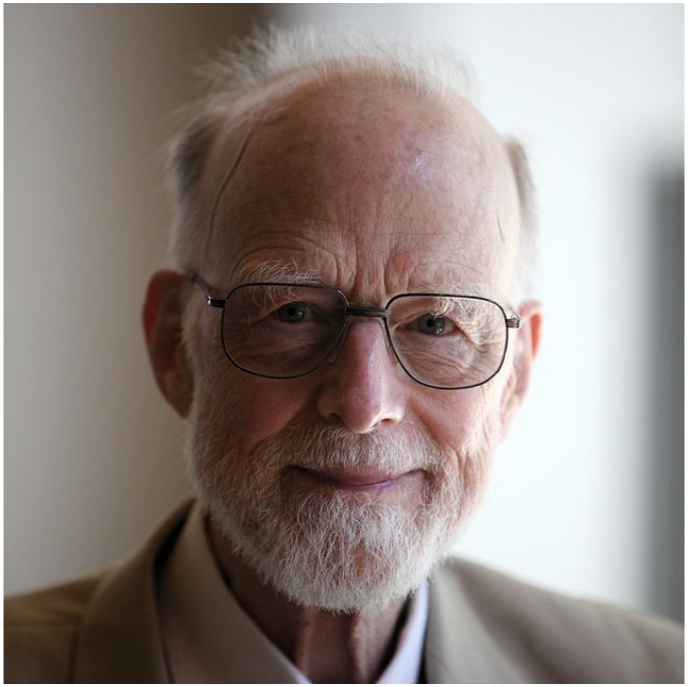
Examples (assume functions of type `Int -> Int`):

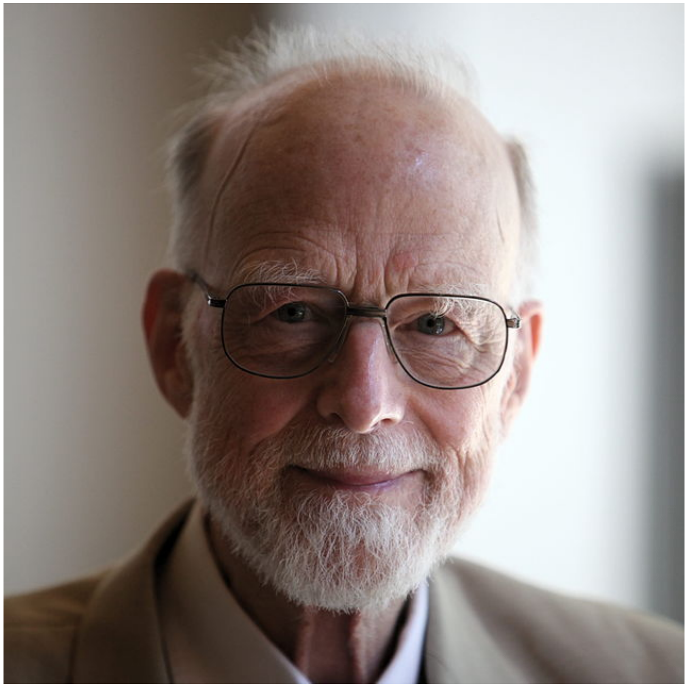
$$\{\text{even}\}(\lambda x \mapsto x+1)\ \{\text{odd}\}.$$

$$\{\text{odd}\}(\lambda x \mapsto x+1)\ \{\text{even}\}.$$

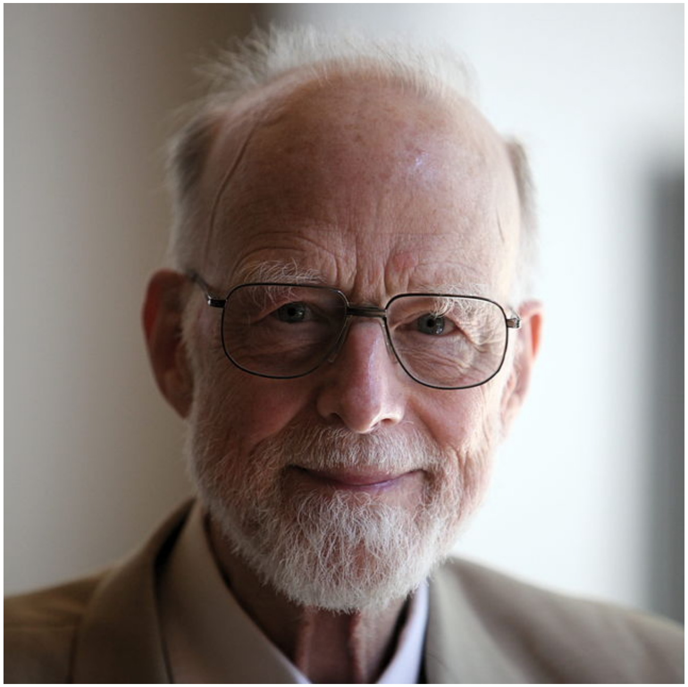
$$\{\top\}(\lambda x \mapsto 2x)\ \{\text{even}\}.$$

$$\{\bot\}(\lambda x \mapsto 2x)\ \{\text{odd}\}.$$



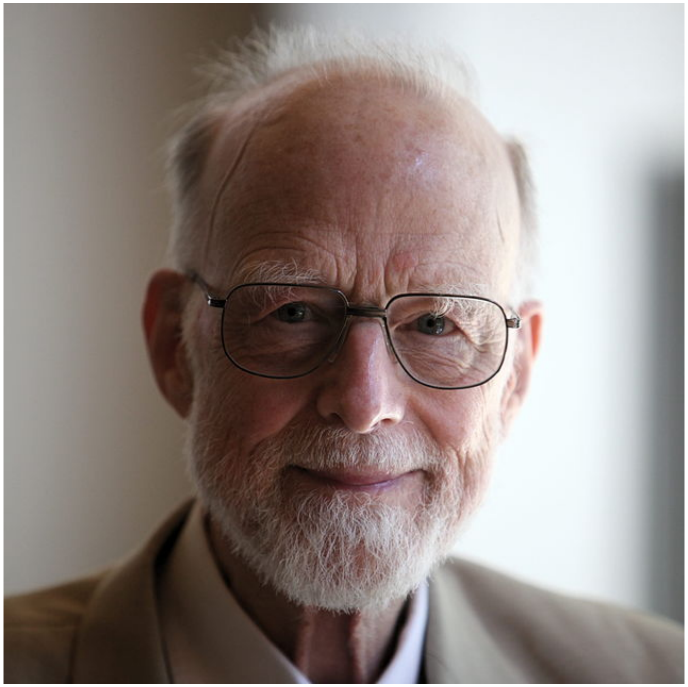


Tony Hoare (born 1934)



Tony Hoare (born 1934)

Hoare logic, QuickSort, Communicating Sequential Processes



Tony Hoare (born 1934)

Hoare logic, QuickSort, Communicating Sequential Processes

Turing award winner (1980)

Precondition Strengthening

$$\frac{p' \models p \qquad \{p\}\ f\ \{q\}}{\{p'\}\ f\ \{q\}}$$

Explanation of

$$p' \models p$$

Property p' is stronger than property p.

Stronger, Weaker

Stronger

$p \models q$

"p is stronger than q"

This means that p rules out more situations than q.

More precisely:

p is stronger than q iff (if and only if) every situation s for which p holds is a situation for which q holds.

Weaker

We say that p is weaker than q in case q is stronger that p.

Implementation

A function of type $a \to a$ (a unary function with arguments and values of the same type) can be tested with test properties of the type $a \to \text{Bool}$.

Define the following predicates on test properties:

```
stronger, weaker :: [a] ->
            (a -> Bool) -> (a -> Bool) -> Bool
stronger xs p q = forall xs (\ x -> p x ==> q x)
weaker   xs p q = stronger xs q p
```

Note the presence of an argument of type `[a]` for the test domain.

The Weakest and the Strongest Property

Use $\top$ for the property that always holds.

This is the weakest property.

Implementation: `\ _ -> True`.

Use $\bot$ for the property that never holds.

This is the strongest property.

Implementation: `\ _ -> False`.

Everything satisfies `\ _ -> True`.

Nothing satisfies `\ _ -> False`.

Negating a Property

```
neg :: (a -> Bool) -> a -> Bool
neg p = \ x -> not (p x)
```

Simpler version

`(not.)` has the same meaning as `neg`.

`(not.)` = `\ p -> not . p` = `\ p x -> not (p x)`

Conjunctions and Disjunctions of Properties

```
infixl 2 .&&.
infixl 2 .||.

(.&&.) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
p .&&. q = \ x -> p x && q x

(.||.) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
p .||. q = \ x -> p x || q x
```

What is the difference between `(&&)` and `(.&&.)`?

What is the difference between `(||)` and `(.||.)`?

Examples

```
*Week2> stronger [0..10] even (even .&&. (>3))
False
*Week2> stronger [0..10] even (even .||. (>3))
True
*Week2> stronger [0..10] (even .&&. (>3)) even
True
*Week2> stronger [0..10] (even .||. (>3)) even
False
```

Further exercises with this: workshop of today.

Importance of Precondition Strengthening for Testing

If you strengthen the requirements on your inputs, your testing procedure gets weaker.

Reason: the set of relevant tests gets smaller.

Note: the precondition specifies the relevant tests.

Preconditions should be as weak as possible (given a function and a postcondition).

Pre- and Postcondition Reasoning: Postcondition Weakening

$$\frac{\{p\}\ f\ \{q\} \qquad q \models q'}{\{p\}\ f\ \{q'\}}$$

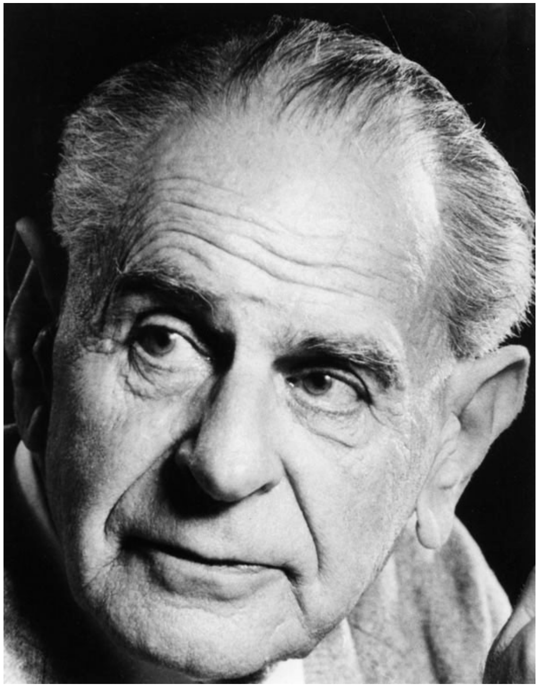
Importance of Postcondition Weakening for Testing

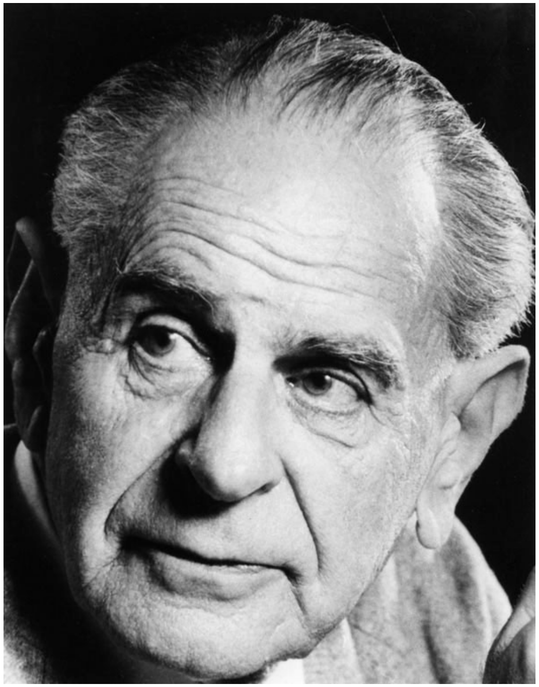
If you weaken the requirements on your outputs in the testing process, your testing procedure gets weaker.

Reason: weakening of the requirements on the outputs increases the number of tests that succeed.

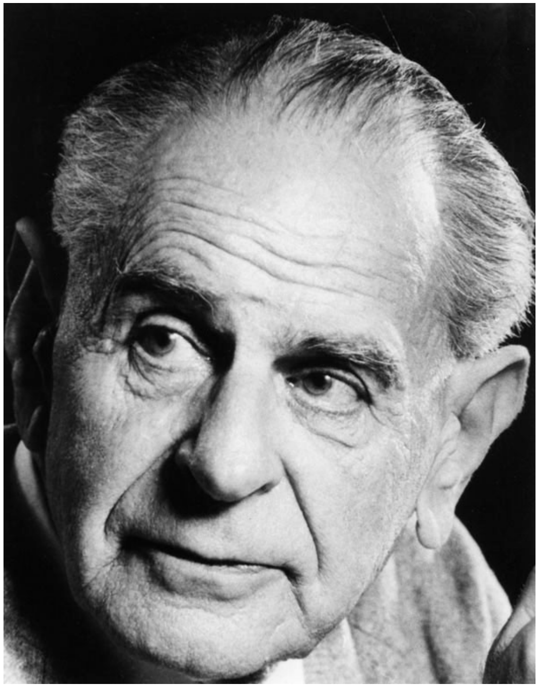
Note: the postcondition specifies the tests that should succeed.

Postconditions should be as strong as possible (given a function and a precondition).



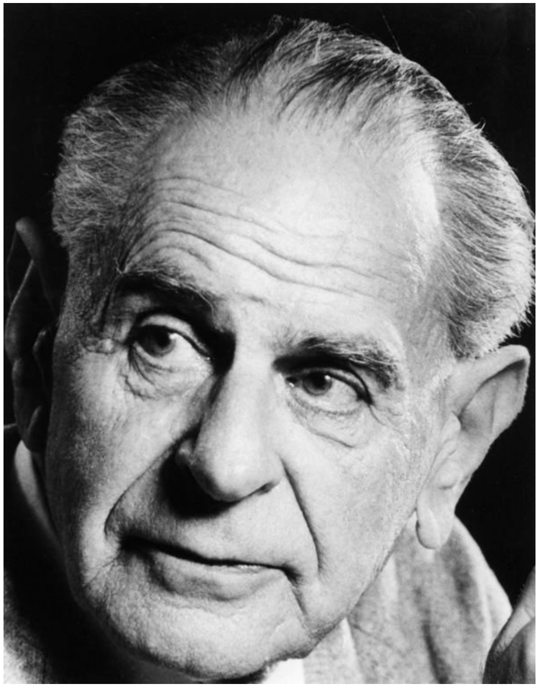


Karl Popper (1902–1994)



Karl Popper (1902–1994)

Falsifiability, Critical Rationalism, Open Society



Karl Popper (1902–1994)

Falsifiability, Critical Rationalism, Open Society

To say something meaningful, say something that can turn out false.

Pre- and Postcondition Composition Rule

$$\frac{\{p\}\ g\ \{q\} \qquad q \models q' \qquad \{q'\}\ f\ \{r\}}{\{p\}\ f \cdot g\ \{r\}}$$

Since $p \models p$ holds for any property p, here is a special case:

$$\frac{\{p\}\ g\ \{q\} \qquad \{q\}\ f\ \{r\}}{\{p\}\ f \cdot g\ \{q\}}$$

These rules look a bit strange because of the notation for function composition, where the function that is applied second is written first.

Function Composition, With Flipped Order

Look again at the definition of function composition:

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

It is useful to flip the order:

```
infixl 2 #

(#) :: (a -> b) -> (b -> c) -> (a -> c)
(#) = flip (.)
```

Pre- and Postcondition Composition Rule Again

$$\frac{\{p\}\ f\ \{q\} \qquad q \models q' \qquad \{q'\}\ g\ \{r\}}{\{p\}\ f\#g\ \{r\}}$$

$$\frac{\{p\}\ f\ \{q\} \qquad \{q\}\ g\ \{r\}}{\{p\}\ f\#g\ \{r\}}$$

Read $f\#g$ as "f followed by g".

Examples:

$$\frac{\{\top\}\ \ x \mapsto 2x\ \{\text{even}\} \qquad \{\text{even}\}\ \ x \mapsto x+1\ \{\text{odd}\}}{\{\top\}\ \ x \mapsto 2x \ \#\ x \mapsto x+1\ \{\text{odd}\}}$$

$$\frac{\{\top\}\ \ x \mapsto x+1\ \{\top\} \qquad \{\top\}\ \ x \mapsto 2x\ \{\text{even}\}}{\{\top\}\ \ x \mapsto x+1 \ \#\ x \mapsto 2x\ \{\text{even}\}}$$

Function Application, with Flipped Order

```
infixl 1 $$

($$) :: a -> (a -> b) -> b
($$) = flip ($)
```

Example:

```
*Week2> 5 $$ succ
6
```

Question: can you work out the type and the definition of `$`?

Question: why is `$` a useful operation?

No Assignment in Pure Functional Programming

What is the difference between $\lambda x \mapsto x + 1$ and $x := x + 1$?

The types are different:

$\lambda x \mapsto x + 1$ is a function.

$x := x+1$ is interpreted in the context of a current memory allocation (an environment), with x naming a memory cell.

$$\begin{array}{ccccc} \cdots & x & y & z & \cdots \\ \cdots & \Box & \Box & \Box & \cdots \end{array}$$

Such an environment (for integers, say), is a function of type $V \rightarrow Int$, where V is a set of variables.

Assignment as Update

Assignment can be viewed as updating the definition of a function.

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (x,y) = \ z -> if x == z then y else f z
```

More specifically, the command $x := x + 1$ is an update operation on an environment.

Assignment as Environment Update

```
type Env = String -> Int
```

To implement variable assignment we need a datatype for expressions, for the assign command assigns an expression to a variable.

```
data Expr = I Int | V String
          | Add Expr Expr
          | Subtr Expr Expr
          | Mult Expr Expr
          deriving (Eq,Show)
```

Evaluation of an expression in an environment

```
eval :: Expr -> Env -> Int
eval (I i) c = i
eval (V name) c = c name
eval (Add e1 e2) c = (eval e1 c) + (eval e2 c)
eval (Subtr e1 e2) c = (eval e1 c) - (eval e2 c)
eval (Mult e1 e2) c = (eval e1 c) * (eval e2 c)
```

Variable Assignment in an Environment

```
assign :: String -> Expr -> Env -> Env
assign var expr c = let
  value = eval expr c
 in
  update c (var,value)
```

An environment is a finite object, so it will yield $\perp$ (undefined) for all but a finite number of variables.

The initial environment is everywhere undefined:

```
initc :: Env
initc = \ _ -> undefined
```

Silly Example

```
example = initc $$
            assign "x" (I 3) #
            assign "x" (Mult (V "x") (V "x")) #
            eval (V "x")
```

```
Week2> :t example
example :: Int
*Week2> example
9
```

The Four Ingredients of Imperative Programming

- Variable Assignment: `<var> := <expr>`
- Conditional Execution:
  `if <bexpr> then <statement1> else <statement2>`
- Sequential Composition: `<statement1> ; <statement2>`
- Iteration: `while <expr> do <statement>`

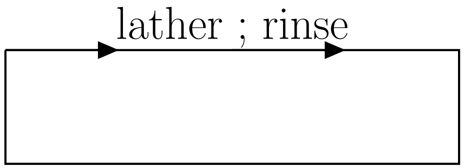
These four ingredients make for a Turing complete programming language.

Turing completeness: a programming language is Turing complete if it is powerful enough to simulate a single taped Turing machine.

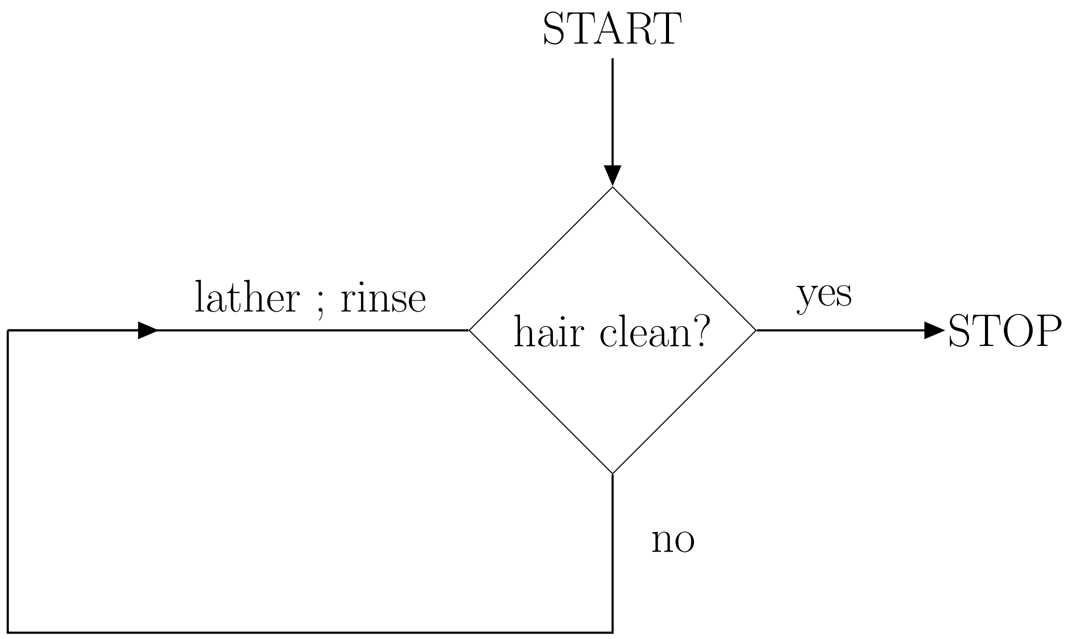
It is believed that such languages can express any function that can be computed by an algorithm.

While Loops

If taken literally, the compound action ‘lather, rinse, repeat’ would look like this:



Repeated actions usually have a stop condition: repeat the lather rinse sequence until your hair is clean. This gives a more sensible interpretation of the repetition instruction:



Written as an algorithm

Hair wash algorithm

- while hair not clean do:
  1. lather;
  2. rinse.

Expressing Iteration as a Pure Function

```
while :: (a -> Bool) -> (a -> a) -> a -> a
while = until . (not.)
```

Famous example:

```
euclid m n = (m,n) $$
   while (\ (x,y) -> x /= y)
         (\ (x,y) -> if x > y then (x-y,y)
                              else (x,y-x)) #
         fst
```

While + Return

Sometimes it is useful to include a function for transforming the result:

```
whiler :: (a -> Bool)
       -> (a -> a) -> (a -> b) -> a -> b
whiler p f r = r . while p f
```

Example

```
euclid2 m n = (m,n) $$
           whiler (\ (x,y) -> x /= y)
                  (\ (x,y) -> if x > y then (x-y,y)
                                       else (x,y-x))
                  fst
```

Random Number Generation

Getting a random integer:

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

```
*Week2> :t getRandomInt
getRandomInt :: Int -> IO Int
*Week2> getRandomInt 20
2
*Week2> getRandomInt 20
11
*Week2> getRandomInt 20
8
```

Generating a List of Integers

Randomly flipping the value of an Int:

```haskell
randomFlip :: Int -> IO Int
randomFlip x = do
  b <- getRandomInt 1
  if b==0 then return x else return (-x)
```

Random integer list:

```haskell
genIntList :: IO [Int]
genIntList = do
  k <- getRandomInt 20
  n <- getRandomInt 10
  getIntL k n

getIntL :: Int -> Int -> IO [Int]
getIntL _ 0 = return []
getIntL k n = do
    x <-  getRandomInt k
    y <- randomFlip x
    xs <- getIntL k (n-1)
    return (y:xs)
```

```
*Week2> genIntList
[-17,-13,1]
```

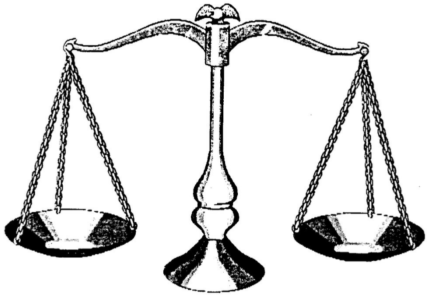
```
*Week2> genIntList
[-13,12,16]
*Week2> genIntList
[0,-3,4,2,-2,0,1,3]
*Week2> genIntList
[-8,5,8,4,5,2,0,-10,10,10]
*Week2> genIntList
[4,-1,-1,17,17,0,-3,8]
*Week2> genIntList
[10,-10,6,9,2,-2,-4]
*Week2> genIntList
[1,5,5,2,2,-2,-3,-9,-13]
```

Appendix: How Much Does a Test Reveal?

You have 27 coins and a balance scale. All coins have the same weight, except for one coin, which is counterfeit: it is lighter than the other coins.

1. How many weighing tests are needed to find the light coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Implementation of a balance scale



A balance scale gives three possible outcomes:

1. Left scale is lighter
2. Left scale is heavier
3. The two scales are in balance.

For this we can use the Haskell datatype `Ordering`, which takes the three values `LT`, `GT`, and `EQ`.

```
data Coin = C Int

w :: Coin -> Float
w (C n) = if n == lighter then 1 - 0.01
          else if n == heavier then 1 + 0.01
          else 1

weight :: [Coin] -> Float
weight = sum . (map w)

balance :: [Coin] -> [Coin] -> Ordering
balance xs ys =
  if weight xs < weight ys then LT
  else if weight xs > weight ys then GT
  else EQ
```

Using the Balance

```
*Week2> balance [C 1] [C 2]
EQ
*Week2> balance [C 1, C 2] [C 3, C 4]
GT
```

Solution

```
*Week2> balance [C i | i <- [1..9]] [ C i | i <- [10..18]]
LT
*Week2> balance [C i | i <- [1..3]] [ C i | i <- [4..6]]
LT
*Week2> balance [C 1] [C 2]
EQ
```

So `C 3` is the counterfeit coin.

Second Testing Challenge

This time you have 3 coins and a balance. Two coins have the same weight, but there is one coin which has different weight from the other coins.

1. How many weighing tests are needed to find the odd coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Third Testing Challenge

This time you have 12 coins and a balance. All coins have the same weight, except for one coin which has different weight from the other coins.

1. How many weighing tests are needed to find the odd coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Some General Questions

1. Using a balance can be viewed as a test with three possible outcomes. If you perform n balance tests, how much information does that give you (at most)?

2. How much information is required to pick out an odd coin (lighter or heavier) from among n other coins? What can you conclude?

3. If you perform n tests, each of which has m possible outcomes, how much information does your testing activity give you?