Conditional Expressions in Functions

We begin our exploration of function-writing techniques with a technique probably familiar to you by now, namely *conditional expressions*. We also look at a variant of conditional expressions that is popular in Haskell functions, namely *guarded equations*.

Conditional Expressions

Conditional expressions are one of the most standard control operations. We check the value of a boolean expression, and choose one of two branches depending on the result. This is done with the standard syntax if <test> then <TrueBranch> else <FalseBranch>.

Note that in Haskell you cannot avoid having the else branch: The expression must evaluate to something one way or another.

> The if–then–else syntax in Haskell is an **expression**: it results in a value. It is in that sense similar to the ternary operator in C or Java: x > 4 ? 3 : 1.

Example of a function that finds the minimum of two numbers, along with some properties it should satisfy:

```
myMin :: Integer -> Integer -> Integer
myMin x y = if x < y then x else y

prop_minAlwaysSmallerThanBoth :: Integer -> Integer -> Bool
prop_minAlwaysSmallerThanBoth x y =    (myMin x y <= x) && (myMin x y <= y)

prop_minAlwaysEqualToOne :: Integer -> Integer -> Bool
prop_minAlwaysEqualToOne x y =    (myMin x y == x) || (myMin x y == y)
```

Save in a file and load, then use quickCheck:

```
quickCheck prop_minAlwaysSmallerThanBoth
quickCheck prop_minAlwaysEqualToOne
```

Guards

A very common practice in Haskell is to use so-called guarded expressions, or **guards**. These are handy when you have more than one condition to test. Conditions are tested one at a time until a True case is found, then that particular path is followed. It is customary to use the special value otherwise which is equal to True as the last case. Here is our myMin function written using guards:

```
myMin :: Integer -> Integer -> Integer
myMin x y | x < y      = x
          | otherwise = y
```

Example: The Collatz Function

The collatz function is defined for natural numbers as follows: If the number is even, divide it by 2. If it is, multiply it by 3 and add 1. For example:

```
collatz 4 = 2
collatz 5 = 16
```

Write a collatz function using guards.

The *Collatz conjecture* is a famous conjecture that says that no matter what number we start with, if we were to apply the collatz function over and over again we eventually end up at 1. This is still an unsolved problem. But we will explore it by writing a function that applies the same function over and over again and records the results, stopping if it ever reaches a prescribed value. We will learn how to write such functions later, but you should be able to follow its logic and understand its type:

```
iter :: Eq t => Int -> (t -> t) -> t -> t -> [t]
iter times f stopAt start
      | times == 0      = []
      | start == stopAt = []
      | otherwise       = nextValue : iter (times - 1) f stopAt nextValue
      where nextValue = f start

testCollatz = iter 1000 collatz 1
```

You can now test different initial numbers like so: testCollatz 51. Try many initial numbers. Does the sequence seem to always reach 1?