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*.

Reading

- Sections 4.1-4.3

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.

As a running example for this section, let us imagine an application that handles people's accounts. In order to avoid using negative numbers when storing account information, each person's account is stored as a pair Account = (Bool, Double), where the boolean determines whether the account has zero or more funds, corresponding to the value True or whether it ows funds, corresponding to the value False. So the number on the second coordinate is always non-negative, and it corresponds to funds in the account in the True case and to funds owed in the False case.

We can now imagine a number of functions we could write in such a setting:

```
-- Giving a new name to this tuple type
type Account = (Bool, Double)
-- Initialize an account given an initial amount (possibly negative).
initAccount :: Double -> Account
-- Deposits a *positive* amount to an account. Returns the "new" account.
deposit :: Double -> Account -> Account
-- Withdraws a *positive* amount from an account. Returns the "new" account.
withdraw :: Double -> Account -> Account
-- Returns whether the account has enough credit for a specific amount of withdrawal.
hasEnoughFunds :: Double -> Account -> Bool
-- Returns whether the account owes funds
doesNotOwe :: Account -> Bool
```

Practice: Before we move on to implement these functions, write in pseudocode how they might go.

Let's think about how we might implement some of these functions:

```
initAccount :: Double -> Account
initAccount amount =
    if amount < 0 then (False, -amount) else (True, amount)
```

```
deposit :: Double -> Account -> Account
deposit amountAdded (hasFunds, amount) =
    if hasFunds
    then (True, amount + amountAdded)
    else initAccount (amountAdded - amount)
```

The above code also exhibits the two different styles when writing "if-then-else" expressions: You can put everything in one line if it is short enough to be readable, or you can vertically align the three keywords.

Practice: Write the withdraw and hasEnoughFunds functions. Write the doesNotOwe function by using the hasEnoughFunds function.

Guarded Expressions

A very common practice in Haskell is to use so-called guarded expressions. 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. For example, here are the initAccount and deposit functions written with guarded expressions:

```
initAccount :: Double -> Account
initAccount amount | amount < 0 = (False, -amount)
                   | otherwise  = (True, amount)

deposit :: Double -> Account -> Account
deposit amountAdded (hasFunds, amount)
        | hasFunds  = (True, amount + amountAdded)
        | otherwise = initAccount (amountAdded - amount)
-- Alternative deposit without using initAccount. Not as nice
deposit :: Double -> Account -> Account
deposit amountAdded (hasFunds, amount)
        | hasFunds        = (True, amount + amountAdded)
        | newBalance < 0 = (False, -newBalance)
        | otherwise       = (True, newBalance)
        where newBalance = amountAdded - amount
```

In all these examples, not the use of the word otherwise. This is simply a synomym for True. Therefore branches with otherwise will always be taken.

Practice: Write the withdraw and hasEnoughFunds functions using guarded expressions instead.

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 guarded expressions.

The *Collatz conjecture* is a famous conjecture that says that no matter what number 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 start numbers like so: testCollatz 51. Try many start numbers. Does the sequence seem to always reach 1?