Chapter 3

Case by Case

In the previous chapter, we used the conditional expression **`if`** ... **`then`** ... **`else`** to define functions whose results depend on their arguments. For some of them we had to nest the conditional expressions one inside another. Programs like this are not terribly easy to read, and expand quickly in size and complexity as the number of cases increases.

Haskell has a nicer way of expressing choices – *pattern matching*. For example, recall our factorial function:

```
factorial :: (Eq a, Num a) ⇒ a → a

factorial n =
  if n == 1 then 1 else n * factorial (n - 1)
```

We can rewrite this using pattern matching:

```
factorial :: (Eq a, Num a) ⇒ a → a

factorial 1 = 1
factorial n = n * factorial (n - 1)
```

We can read this as "See if the argument matches the pattern `1`. If it does, just return `1`. If not, see if it matches the pattern `n`. If it does, the result is `n * factorial (n - 1)`." Patterns like `n` are special – they match anything and give it a name. Remember our `isVowel` function from the previous chapter?

```
isVowel :: Char → Bool

isVowel c =
  c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
```

Here is how to write it using pattern matching:

```
isVowel :: Char → Bool

isVowel 'a' = True
isVowel 'e' = True
isVowel 'i' = True
isVowel 'o' = True
isVowel 'u' = True
isVowel _ = False
```

The special pattern _ matches anything. If we miss out one or more cases – for example leaving out the final case, Haskell can warn us:

```
<interactive> warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'isVowel':
        Patterns not matched:
            p where p is not one of {'u', 'o', 'i', 'e', 'a'}
```

To enable this behaviour, you must start Haskell by writing `ghci -Wincomplete-patterns` instead of just `ghci`. Writing `ghci -Wall` enables all warnings. Haskell does not reject the program outright, because there may be legitimate reasons to miss cases out, but for now we will make sure all our pattern matches are exhaustive. Finally, let us rewrite Euclid's Algorithm from the previous chapter:

```
gcd' :: Integral a ⇒ a → a → a

gcd' a b =
  if b == 0 then a else gcd' b (a `rem` b)
```

Now in pattern matching style:

```
gcd' :: Integral a ⇒ a → a → a

gcd' a 0 = a
gcd' a b = gcd' b (a `rem` b)
```

We use pattern matching whenever it is easier to read and understand than **`if`** … **`then`** … **`else`** expressions.

What about this **Integral** typeclass? We did not try `:type` on the `gcd'` function in the last chapter, so we did not see this. A type of number which is an **Integral** has an additional property to one which is merely a **Num**, which is that whole-number division and remainder operations work on it. Since every thing which is an **Integral** is also a **Num**, we do not see (**Num** a, **Integral** a), but just **Integral** a in the type.

Sometimes we need more than just a pattern to decide which case to choose in a pattern match. For example, in `gcd'` above, we only needed to distinguish between `0` and any other value of `b`. Consider, though, the function to determine the sign of a number, producing -1 for all numbers less than zero, 0 for just the number zero, and 1 for all numbers above zero:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x =
  if x < 0 then -1 else if x > 0 then 1 else 0
```

We cannot rewrite this using a pattern match with three cases. Haskell has a facility called *guarded equations* to help us (each line in our pattern matched functions can also be called an equation). A *guard* is an extra check to decide if a case of a pattern match is taken based upon some condition, for example `x < 0`. Here is our sign function written using guarded equations:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x | x < 0     = -1
       | x > 0     = 1
       | otherwise = 0
```

There is no need to line up the equals signs vertically, but we do so to make it easier to read. The cases are considered one after another, just like when using pattern matching, and the first case which matches the guard is taken. The **`otherwise`** guard matches anything, so it comes last. We use an **`otherwise`** case to make sure every possibility is handled. We can read the `|` symbol as "when". A function can be defined using multiple equations, each of which has multiple guarded parts.

The layout rule

We have mentioned indentation, noting that Haskell is particular about it. Indeed, programs will not be accepted unless they are properly indented:

```
GHCi:
Prelude> :{
Prelude| sign x =
Prelude| if x < 0 then -1 else if x > 0 then 1 else 0
Prelude| :}

<interactive> error:
    parse error (possibly incorrect indentation or mismatched brackets)
```

Haskell is telling us that it cannot work out what we mean. Since the **`if`** ... **`then`** ... **`else`** ... expression is part of the `sign` function, it must be indented further than the beginning of the whole `sign` expression. This applies at all times – even when the start of the whole expression is itself indented. In the case of **`if`** ... **`then`** ... **`else`** ... itself, it is in fact permitted not to indent:

```
GHCi:
Prelude> :{
Prelude| if 1 < 0
Prelude| then 2
Prelude| else 3
Prelude| :}
3
```

However, we shall often do so, when it is easier to read:

```
GHCi:
Prelude> :{
Prelude| if 1 < 0
Prelude|   then 2
Prelude|   else 3
Prelude| :}
3
```

Consider again our `sign` function:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x | x < 0     = -1
       | x > 0     = 1
       | otherwise = 0
```

We have already mentioned that lining up the equals signs is not necessary. However, we must always indent the cases. Here, we start the cases on the next line:

```
Prelude| sign x
Prelude|   | x < 0 = -1
Prelude|   | x > 0 = 1
Prelude|   | otherwise = 0
Prelude| :}
```

The layout rule is not complicated, but it can be frustrating to the beginner, especially when the error message is not clear.

Questions

1. Rewrite the `not'` function from the previous chapter in pattern matching style.
2. Use pattern matching to write a recursive function `sumMatch` which, given a positive integer n, returns the sum of all the integers from 1 to n.
3. Use pattern matching to write a function which, given two numbers x and n, computes x^n.
4. For each of the previous three questions, comment on whether you think it is easier to read the function with or without pattern matching. How might you expect this to change if the functions were much larger? Write each using guarded equations too.
5. Use guarded equations to write a function which categorises characters into three kinds: kind 0 for the lowercase letters `a`...`z`, kind 1 for the uppercase letters `a`...`z`, and kind 2 for everything else.
6. Experiment with the layout of the function definitions in this and the previous chapter. Which kinds of layout are allowed by Haskell? Which of the allowed layouts are aesthetically pleasing, or easy to read? Do any of your layouts make the program harder to change?