```
concat :: [[a]] -> [a]
```

that concatenates a list of lists of things together. Again, it doesn't matter what the 'thing' is as far as concatenation is concerned, which is why there is an `a` in the type signature.

Now we can define

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

The definition of `commonWords` is given as a pipeline of eight component functions glued together by functional composition. Not every problem can be decomposed into component tasks in quite such a straightforward manner, but when it can, the resulting program is simple, attractive and effective.

Notice how the process of decomposing the problem was governed by the declared types of the subsidiary functions. Lesson Two (Lesson One being the importance of functional composition) is that deciding on the type of a function is the very first step in finding a suitable definition of the function.

We said above that we were going to write a *program* for the common words problem. What we actually did was to write a functional definition of `commonWords`, using subsidiary definitions that we either can construct ourselves or else import from a suitable Haskell library. A list of definitions is called a *script*, so what we constructed was a script. The order in which the functions are presented in a script is not important. We could place the definition of `commonWords` first, and then define the subsidiary functions, or else define all these functions first, and end up with the definition of the main function of interest. In other words we can tell the story of the script in any order we choose. We will see how to compute with scripts later on.

1.4 Example: numbers into words

Here is another example, one for which we will provide a complete solution. The example demonstrates another fundamental aspect of problem solving, namely that a good way to solve a tricky problem is to first simplify the problem and then see how to solve the simpler problem.

Sometimes we need to write numbers as words. For instance

```
convert 308000 = "three hundred and eight thousand"
convert 369027 = "three hundred and sixty-nine thousand and
                  twenty-seven"
convert 369401 = "three hundred and sixty-nine thousand
                  four hundred and one"
```

Our aim is to design a function

```
convert :: Int -> String
```

that, given a nonnegative number less than one million, returns a string that represents the number in words. As we said above, `String` is a predeclared type synonym in Haskell for `[Char]`.

We will need the names of the component numbers. One way is to give these as three lists of strings:

```
> units, teens, tens :: [String]
> units = ["zero","one","two","three","four","five",
>          "six","seven","eight","nine"]
> teens = ["ten","eleven","twelve","thirteen","fourteen",
>          "fifteen","sixteen","seventeen","eighteen",
>          "nineteen"]
> tens  = ["twenty","thirty","forty","fifty","sixty",
>          "seventy","eighty","ninety"]
```

Oh, what is the > character doing at the beginning of each line above? The answer is that, in a script, it indicates a line of Haskell code, not a line of comment. In Haskell, a file ending with the suffix `.lhs` is called a *Literate Haskell Script* and the convention is that every line in such a script is interpreted as a comment unless it begins with a > sign, when it is interpreted as a line of program. Program lines are not allowed next to comments, so there has to be at least one blank line separating the two. In fact, the whole chapter you are now reading forms a legitimate `.lhs` file, one that can be loaded into a Haskell system and interacted with. We won't carry on with this convention in subsequent chapters (apart from anything else, it would force us to use different names for each version of a function that we may want to define) but the present chapter does illustrate *literate* programming in which we can present and discuss the definitions of functions in any order we wish.

Returning to the task in hand, a good way to tackle tricky problems is to solve a simpler problem first. The simplest version of our problem is when the given number n contains only one digit, so $0 \leq n < 10$. Let `convert1` deal with this version. We can immediately define

```
> convert1 :: Int -> String
> convert1 n = units!!n
```

This definition uses the list-indexing operation `(!!)`. Given a list `xs` and an index `n`, the expression `xs!!n` returns the element of `xs` at position `n`, counting from 0. In particular, `units!!0 = "zero"`. And, yes, `units!!10` is undefined because `units` contains just ten elements, indexed from 0 to 9. In general, the functions we define in a script are *partial* functions that may not return well-defined results for each argument.

The next simplest version of the problem is when the number n has up to two digits, so $0 \leq n < 100$. Let `convert2` deal with this case. We will need to know what the digits are, so we first define

```
> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)
```

The number `div n k` is the whole number of times `k` divides into `n`, and `mod n k` is the remainder. We can also write

```
    digits2 n = (n `div` 10, n `mod` 10)
```

The operators `` `div` `` and `` `mod` `` are infix versions of `div` and `mod`, that is, they come between their two arguments rather than before them. This device is useful for improving readability. For instance a mathematician would write $x \operatorname{div} y$ and $x \bmod y$ for these expressions. Note that the back-quote symbol `` ` `` is different from the single quote symbol `'` used for describing individual characters.

Now we can define

```
> convert2 :: Int -> String
> convert2 = combine2 . digits2
```

The definition of `combine2` uses the Haskell syntax for *guarded equations*:

```
> combine2 :: (Int,Int) -> String
> combine2 (t,u)
>   | t==0         = units!!u
>   | t==1         = teens!!u
>   | 2<=t && u==0 = tens!!(t-2)
>   | 2<=t && u/=0 = tens!!(t-2) ++ "-" ++ units!!u
```

To understand this code you need to know that the Haskell symbols for equality and comparison tests are as follows:

`==`	(equals to)
`/=`	(not equals to)
`<=`	(less than or equal to)

These functions have well-defined types that we will give later on.

You also need to know that the conjunction of two tests is denoted by `&&`. Thus `a && b` returns the boolean value `True` if both `a` and `b` do, and `False` otherwise. In fact

```
(&&) :: Bool -> Bool -> Bool
```

The type `Bool` will be described in more detail in the following chapter.

Finally, `(++)` denotes the operation of concatenating two lists. It doesn't matter what the type of the list elements is, so

```
(++) :: [a] -> [a] -> [a]
```

For example, in the equation

```
[sin,cos] ++ [tan] = [sin,cos,tan]
```

we are concatenating two lists of functions (each of type `Float -> Float`), while in

```
"sin cos" ++ " tan" = "sin cos tan"
```

we are concatenating two lists of characters.

The definition of `combine2` is arrived at by carefully considering all the possible cases that can arise. A little reflection shows that there are three main cases, namely when the tens part `t` is 0, 1 or greater than 1. In the first two cases we can give the answer immediately, but the third case has to be divided into two subcases, namely when the units part `u` is 0 or not 0. The order in which we write the cases, that is, the order of the individual guarded equations, is unimportant as the guards are disjoint from one another (that is, no two guards can be true) and together they cover all cases.

We could also have written

```
combine2 :: (Int,Int) -> String
combine2 (t,u)
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
```

but now the order in which we write the equations is crucial. The guards are evaluated from top to bottom, taking the right-hand side corresponding to the first guard that evaluates to `True`. The identifier `otherwise` is just a synonym for `True`, so the last clause captures all the remaining cases.

There is yet another way of writing `convert2`:

```
convert2 :: Int -> String
convert2 n
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
  where (t,u) = (n `div` 10, n `mod` 10)
```

This makes use of a `where` *clause*. Such a clause introduces a *local* definition or definitions whose *context* or *scope* is the whole of the right-hand side of the definition of `convert2`. Such clauses are very useful in structuring definitions and making them more readable. In the present example, the `where` clause obviates the need for an explicit definition of `digits2`.

That was reasonably easy, so now let us consider `convert3` which takes a number n in the range $0 \leq n < 1000$, so n has up to three digits. The definition is

```
> convert3 :: Int -> String
> convert3 n
>   | h==0      = convert2 t
>   | n==0      = units!!h ++ " hundred"
>   | otherwise = units!!h ++ " hundred and " ++ convert2 t
>   where (h,t) = (n `div` 100, n `mod` 100)
```

We break up the number in this way because we can make use of `convert2` for numbers that are less than 100.

Now suppose n lies in the range $0 \leq n < 1,000,000$, so n can have up to six digits. Following exactly the same pattern as before, we can define

```
> convert6 :: Int -> String
> convert6 n
>   | m==0      = convert3 h
>   | h==0      = convert3 m ++ " thousand"
>   | otherwise = convert3 m ++ " thousand" ++ link h ++
>                 convert3 h
>   where (m,h) = (n `div` 1000,n `mod` 1000)
```

There will be a connecting word ‘and’ between the words for m and h just in the case that $0 < m$ and $0 < h < 100$. Thus

```
> link :: Int -> String
> link h = if h < 100 then " and " else " "
```

This definition makes use of a conditional expression

```
    if <test> then <expr1> else <expr2>
```

We could also have used guarded equations:

```
    link h | h < 100   = " and "
           | otherwise = " "
```

Sometimes one is more readable, sometimes the other. The names `if`, `then` and `else`, along with some others, are *reserved words* in Haskell, which means that we cannot use them as names for things we want to define.

Notice how the definition of `convert6` has been constructed in terms of the simpler function `convert3`, which in turn has been defined in terms of the even simpler function `convert2`. That is often the way with function definitions. In this example consideration of the simpler cases is not wasted because these simple cases can be used in the final definition.

One more thing: we have now named the function we are after as `convert6`, but we started off by saying the name should be `convert`. No problem:

```
> convert :: Int -> String
> convert = convert6
```

What we would like to do now is actually use the computer to apply `convert` to some arguments. How?

1.5 The Haskell Platform

If you visit the site `www.haskell.org`, you will see how to download *The Haskell Platform*. This is a large collection of tools and packages that can be used to run Haskell scripts. The platform comes in three versions, one for each of Windows, Mac and Linux. We deal only with the Windows version, the others being similar.

One of the tools is an interactive calculator, called GHCi. This is short for *Glasgow Haskell Compiler Interpreter*. The calculator is available as a Windows system called WinGHCi. If you open this window, you will get something like