2 Definitions

2.1 A problem to be solved

The example in this lecture is to design a function

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

that translates a number $0 \leqslant n < 1\,000\,000$ into a string which names that number in English. The type *Int* is the type of limited precision integers, and *String* is a synonym for $[Char]$.

2.2 Numbers less than 10

A good approach to a daunting problem is to solve smaller problems first. The most straightforward way to convert a small number, $0 \leqslant n < 10$ is to deal with each value as a special case, giving an equation for each value:

```
> name :: Int -> String
> name 0 = "zero"
> name 1 = "one"
> name 2 = "two"
> name 3 = "three"
> name 4 = "four"
> name 5 = "five"
> name 6 = "six"
> name 7 = "seven"
> name 8 = "eight"
> name 9 = "nine"
```

but perhaps it would be tidier to look up the answer in a list:

```
> units :: Int -> String
> units u = unitStrings!!u

> unitStrings :: [String]
> unitStrings = [ "zero", "one", "two",   "three", "four",
>                 "five", "six", "seven", "eight", "nine" ]
```

The operator

```
(!!) :: [a] -> Int -> a
```

selects an item from a list according to its position, starting to count from position zero.

2.3 Numbers less than 100

There is a pattern (in English, a slightly complicated one) to the names of two digit numbers which it is worth exploiting.

If $0 \leqslant n < 100$, the tens digit is *div n* 10 and the units digit is *mod n* 10.

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

Perhaps more usually one would write

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

The ` marks (called backquotes) convert a name like div into an infix operator `div`. (This is a different character from the single quote ' which marks a character constant like 'x'.)

There is a standard function *divMod* which satisfies

```
    p `divMod` q = (p `div` q, p `mod` q)
```

but which only performs one division operation to produce both results, and so is more efficient when as here both results are needed, so we could write

```
> digits2 n = n `divMod` 10
```

but we choose not to here.

Now we can define

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

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

where $(+\!\!+)$, pronounced 'cat', concatenates two lists: in this case two lists of characters. The guarded equations in *combine2* are all part of one equation defining the function according to which of conditions are true.

The guards are written using tests: (==) for equality, (/=) for inequality, and (<=) for no-more-than, all of which return a value of type *Bool*. This type has two values *True* and *False*, and the && operator (read *and*) returns *True* exactly when both arguments are *True*.

As written only one of the guards is ever true for a given (t, u), but we could have written

```
combine2 (t,u)
  | t==0      = units u
  | t==1      = teens u
  | u==0      = tens t
  | otherwise = tens t ++ "-" ++ units u
```

where *otherwise* is a predefined constant equal to *True*. This is because the guards are tested in order from the top to the bottom, and the first *True* guard wins. In this case, the order of the guarded equations matters.

The words for tens and units come from

```
> teens, tens:: Int -> String
> teens u = teenStrings!!u
> tens t  = tenStrings!!(t-2)

> teenStrings :: [String]
> teenStrings = [ "ten",      "eleven",  "twelve", "thirteen",
>                 "fourteen", "fifteen", "sixteen", "seventeen",
>                 "eighteen", "nineteen" ]

> tenStrings :: [String]
> tenStrings = [ "twenty", "thirty",  "forty",  "fifty",
>                "sixty",  "seventy", "eighty", "ninety" ]
```

2.4 Numbers less than 1 000 and less than 1 000 000

This scheme extends to numbers $0 \leqslant n < 1\,000$, with up to three digits, which can be treated as a number of thousands followed by a two digit number.

```
> digits3 :: Int -> (Int,Int)
> digits3 n = (n `div` 100, n `mod` 100)

> convert3 :: Int -> String
> convert3 = combine3 . digits3

> combine3 :: (Int,Int) -> String
> combine3 (h,n)
>   | h==0      = convert2 n
>   | n==0      = units h ++ " hundred"
>   | otherwise = units h ++ " hundred and " ++ convert2 n
```

and finally to $0 \leqslant n < 1\,000\,000$

```
> digits6 :: Int -> (Int,Int)
> digits6 n = (n `div` 1000, n `mod` 1000)
> convert6 :: Int -> String
> convert6 = combine6 . digits6

> combine6 :: (Int,Int) -> String
> combine6 (m,n)
>   | m==0      = convert3 n
>   | n==0      = convert3 m ++ " thousand"
>   | otherwise = convert3 m ++ " thousand" ++ link n ++ convert3 n
```

The function *link* inserts an *and* into the answer exactly when there are thousands, but there are no hundreds:

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

(This is the sort of thing that makes natural language so messy.)

The definition might have been given with guarded equations, but here we illustrate the Haskell conditional expression

`if` *test* `then` *truecase* `else` *falsecase*

Finally

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

2.5 The design pattern

The crucial step in this design is to build each of the numbered *convert* functions from a smaller one. In particular the decision to code *convert3* by using *convert2*. An alternative design would separate a three digit number into three digits and deal with more different cases.

2.6 Local definitions

Sometimes it is useful to be able to give a name to something without that name being available all over the program. Haskell provides a form of expression for this.

```
> f1 n = let phi    = (1 + root5)/2
>            phibar = -1/phi
>            root5  = sqrt 5
>         in (phi^n-phibar^n)/root5
```

Let expressions can be used anywhere where an expression is needed, and the list of local definitions can include any legal defining equation that could have apperared at the top level. (So in particular you can define local functions.)

However let expressions are realtively rarely used. Much more common (because it is in general much more natural to read) is the where clause

```
> f2 n = (phi^n-phibar^n)/root5
>    where phi    = (1 + root5)/2
>          phibar = -1/phi
>          root5  = sqrt 5
```

which qualifies a defining equation. (The idea of a where clause in a program appears to have started with Christopher Strachey.)

For clarity: a where clause does not qualify the right-hand side of a definition, it qualifies the equation. A where clause cannot be added to anything other than an equation, though of course where clauses can be nested by adding them to equations in an enclosing where clause.

```
> f3 n = (phi^n-phibar^n)/(phi-phibar)
>        where phi = (1 + root5)/2
>                  where root5 = sqrt 5
>              phibar = -1/phi
```

2.7 Offside rule

This last definition illustrates the usefulness of the Haskell offside rule. Equations start in a particular column, and the whole of the rest of the equation (including, if it has any, its where clauses) have to appear to the right of that. The next line which starts in the same column, or to the left, does not continue this equation.

It is much easier in practice than that explanation makes it seem. The layout of Haskell programs almost always natrally follows the structure, so having to obey the offside rule naturally ecourages good layout.

Haskell notionally uses curly braces { and } and semicolons to outline the structure of code, so

```
let { two = 1+1; three = 1+two } in two + three
```

although the braces and semicolons are almost never used by human programmers. Similarly

```
> five = two + three where { two = 1+1 ; three = 1+two }
```

This punctuation is almost always left out, using layout and the offside rule to make the structure obvious.

The offside rule also applies to other constructs which will appear later.

Reminder: pre-defined names met so far

The only keywords met so far have been `else`, `if`, `in`, `let`, `then` , `where`. Everything else is a predefined name

```
(.)       :: (b -> c) -> (a -> b) -> (a -> c)

map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
concat    :: [[a]] -> [a]

head      :: [a] -> a
reverse   :: [a] -> [a]

length    :: [a] -> Int
take      :: Int -> [a] -> [a]
(!!)      :: [a] -> Int -> a

words     :: String -> [String]

otherwise :: Bool
(&&)      :: Bool -> Bool -> Bool

show      :: Show a => a -> String

div       :: Integral a => a -> a -> a
mod       :: Integral a => a -> a -> a
divMod    :: Integral a => a -> a -> (a, a)
sqrt      :: Floating a => a -> a

(==)      :: Eq a => a -> a -> Bool
(/=)      :: Eq a => a -> a -> Bool

(<)       :: Ord a => a -> a -> Bool
(<=)      :: Ord a => a -> a -> Bool
(>=)      :: Ord a => a -> a -> Bool
(>)       :: Ord a => a -> a -> Bool
```

additionally, for import from `Data.List`

```
group     :: Eq a =>  [a] -> [[a]]
sort      :: Ord a => [a] -> [a]
```

and for import from `Data.Char`

```
isLower   :: Char -> Bool
isUpper   :: Char -> Bool
toLower   :: Char -> Char
```