Function definition

We saw yesterday that functions are defined in Haskell as (sets of) equations:

\[
\begin{align*}
\text{sum} \; [] & = 0 \\
\text{sum} \; (n:ns) & = n + \text{sum} \; ns
\end{align*}
\]

Let’s look in more detail at how these are written, and how the system selects which equation applies in any given case.

Definition by cases

Often we want to have different equations for different cases. Mathematically we sometimes write something like this:

\[
\text{signum} \; x = \begin{cases} 
0 & \text{if } x = 0 \\
-1 & \text{if } x < 0 \\
1 & \text{if } x > 0 
\end{cases}
\]

This indicates that there are three different cases in the definition of \textit{signum}, and gives a rule for choosing which case should applied in any particular application.

Cases by pattern matching

When there is more than one equation defining a function there must be some way to select between them. One particularly convenient way to do this is \textit{pattern matching}:

\[
\begin{align*}
\text{count} \; 0 & = 0 \\
\text{count} \; 1 & = 1 \\
\text{count} \; n & = n-2
\end{align*}
\]

These three equations give a definition for a function. When the function is applied to an integer Haskell selects one of the equations by matching the actual parameter against the patterns.
Cases by pattern matching

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two Bool arguments like this:

and True True = True
and _ _ = False

The special pattern ";" will match any value without binding it to a name. It is usually used to indicate that the value is not needed on the right-hand side.

Partial functions

We can select equations using quite complex patterns. This set is non-exhaustive, and so the sum3 function is partial.

\[
\begin{align*}
\text{sum3} \; [] & = 0 \\
\text{sum3} \; [x] & = x \\
\text{sum3} \; [x,y] & = x+y \\
\text{sum3} \; [x,y,z] & = x+y+z
\end{align*}
\]

It is an error to apply a partial function outside the domain it is defined for (division is another example of a partial function).

Pattern matching on structures

Here is a (partial) function from the Prelude:

\[
\begin{align*}
> \text{head} \; [1,2,3] \\
1
\end{align*}
\]

It provides the first element of a list. We can define it using pattern matching:

\[
\text{head} \; (x:xs) = x
\]

The pattern in this case is based on something called a constructor.

Pattern matching on structures

The pattern match relies on the fact that lists are built out of individual elements using ";:" (“cons”\(^1\)). It’s only for convenience that we use the comma-list notation.

\[
\begin{align*}
> 1:[] \\
[1] \\
> 2:3:[] \\
[2,3] \\
> 1:[2,3] \\
[1,2,3]
\end{align*}
\]

So in the pattern match for \text{head},

\[
\text{head} \; (x:xs) = x
\]

an argument like \([1,2,3]\) would result in \(x\) being associated with 1 and \(xs\) being associated with \([2,3]\)

\(^1\)pron. “cons”
Cases by guarded alternatives

Another way to make choices is to use guarded alternatives (this looks rather more like the mathematical style we saw earlier):

\[
\text{signum } x \mid x < 0 = -1 \\
\mid x == 0 = 0 \\
\mid x > 0 = 1
\]

The guards are boolean expressions. If a guard is true the corresponding right-hand side will be selected as the definition for the function.

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the last alternative. So the definition above could have been written like this:

\[
\text{signum } x \mid x < 0 = -1 \\
\mid x == 0 = 0 \\
\mid \text{True} = 1
\]

Recursive functions

Many important computations can be expressed using the parts of Haskell that we have seen. List comprehensions are sufficient for some operations, such as applying a function to each item in a list (“mapping”), or forming a sequence by selecting some elements from an existing one (“filtering”). But not all computations can be expressed by combining these ideas.

We need some way to express general patterns of repetition. In Haskell the mechanism for this is recursion.

\[
\text{factorial } 0 = 1 \\
\text{factorial } n = n \times \text{factorial } (n-1)
\]

\[
\text{productof } [] = 1 \\
\text{productof } (x:xs) = x \times \text{productof } xs
\]

More examples of recursion

A function to take the first \( n \) elements of a list:

\[
> \text{takesome } 4 \ "abcdefg\h" \\
"abcd"
\]

\[
\text{takesome } 0 \ _ = [] \\
\text{takesome } _ \ [] = [] \\
\text{takesome } n \ (x:xs) = x : (\text{takesome } (n-1) \ xs)
\]

The quicksort algorithm has a particularly elegant expression in Haskell, which we saw last time (here’s it is written slightly differently):

\[
\text{qsort } [] = [] \\
\text{qsort } (x:xs) = \text{lpartition ++ [x]} ++ \text{rpartition} \\
\text{where lpartition = qsort } [ a \mid a \leftarrow xs, a \leq x ] \\
\text{rpartition = qsort } [ a \mid a \leftarrow xs, a > x ]
\]

Cases by guarded alternatives

We can use guards to select special cases in functions. This function is True when the year number is a leap year:

\[
\text{leapyear } y \mid \mod y 100 == 0 = \mod y 400 == 0 \\
\mid \text{True} = \mod y 4 == 0
\]

For readability the name otherwise is allowed as a synonym for True:

\[
\text{leapyear } y \mid y \ 'mod' \ 100 == 0 = y \ 'mod' \ 400 == 0 \\
\mid \text{otherwise} = y \ 'mod' \ 4 == 0
\]

In Haskell any function of two arguments may be written infix if it is surrounded by backquotes, which is why ‘mod’ is OK.

Guards and patterns can be combined:

\[
\text{startswith } _ \ [] = \text{False} \\
\text{startswith } c \ (x:xs) \mid x == c = \text{True} \\
\mid \text{otherwise} = \text{False}
\]
Local definitions

We used two new things in that definition of quicksort: The phrase “where” is followed by one or more function definitions which are local to the containing function. Because they are local the parameters to the containing function are captured and are in scope. In Haskell indentation is significant to meaning. The where clause must be indented deeper than the = of the containing function.

qsort [] = []
qsort (x:xs) = lpartition ++ [x] ++ rpartition
    where lpartition = qsort [ a | a <- xs, a <= x ]
          rpartition = qsort [ a | a <- xs, a > x ]

List comprehensions

We also used a concise way to specify lists, called a list comprehension. The syntax of this is based on the way sets are described in mathematics: \{ x | x \in \mathbb{Z}, x \text{ is even} \}

A list comprehension includes:

- An expression (c, in this case) from which to build the list
- A generator which indicates the set from which the values will be drawn. (c <- "characters")
- Some (optional) guards which correspond to the qualifiers in the mathematics expression. In this case the guard says that only characters which are non-blank will be included. The symbol /= means “is not equal to”, and is equivalent to writing not (c==' ').

qsort [] = []
qsort (x:xs) = lpartition ++ [x] ++ rpartition
    where lpartition = qsort [ a | a <- xs, a <= x ]
          rpartition = qsort [ a | a <- xs, a > x ]

Another example: Matrix transposition

A matrix can be defined using lists of lists (I've spread the definition over more than one line for clarity, but you don't have to do this):

> transpose [ [1,2,3],
          [4,5,6],
          [7,8,9] ]
> transpose [ [1,4,7],
          [2,5,8],
          [3,6,9] ]

transpose [] = []
transpose (xs : xss) = transpose xss
transpose ((x:xs):xss) = row : (transpose rest)
    where row = x : [head l | l <- xss]
          rest = xs : [tail l | l <- xss]