Chapter 4

Lists

Lists are the workhorses of functional programming. They can be used to fetch and carry data from one function to another; they can be taken apart, rearranged and combined with other lists to make new lists. Lists of numbers can be summed and multiplied; lists of characters can be read and printed; and so on. The list of useful operations on lists is a long one. This chapter describes some of the operations that occur most frequently, though one particularly important class will be introduced only in Chapter 6.

4.1 List notation

As we have seen, the type `[a]` denotes lists of elements of type `a`. The empty list is denoted by `[]`. We can have lists over any type but we cannot mix different types in the same list. As examples,

```
[undefined,undefined] :: [a]
[sin,cos,tan]         :: Floating a => [a -> a]
[[1,2,3],[4,5]]       :: Num a => [[a]]
["tea","for",2]          not valid
```

List notation, such as `[1,2,3]`, is in fact an abbreviation for a more basic form

```
1:2:3:[]
```

The operator `(:) :: a -> [a] -> [a]`, pronounced 'cons', is a constructor for lists. It associates to the right so there is no need for parentheses in the above expression. It has no associated definition, which is why it is a constructor. In other words, there are no rules for simplifying an expression such as `1:2:[]`. The

operator `(:)` is non-strict in both arguments – more precisely, it is non-strict and returns a non-strict function. The expression

```
undefined : undefined
```

may not be very interesting, but we do know it is not the empty list. In fact, that is the only thing we do know about it. Note that the two occurrences of `undefined` have different types in this expression.

The empty list `[]` is also a constructor. Lists can be introduced as a Haskell data type with the declaration

```
data List a = Nil | Cons a (List a)
```

The only difference is that `List a` is written `[a]`, `Nil` is written `[]` and `Cons` is written `(:)`.

According to this declaration, every list of type `[a]` takes one of three forms:

- The undefined list `undefined :: [a]`;
- The empty list `[] :: [a]`;
- A list of the form `x:xs` where `x :: a` and `xs :: [a]`.

As a result there are three kinds of list:

- A *finite* list, which is built from `(:)` and `[]`; for example, `1:2:3:[]`
- A *partial* list, which is built from `(:)` and `undefined`; for example, the list `filter (<4) [1..]` is the partial list `1:2:3:undefined`. We know there is no integer after 3 that is less than 4, but Haskell is an evaluator, not a theorem prover, so it ploughs away without success looking for more answers.
- An *infinite* list, which is built from `(:)` alone; for example, `[1..]` is the infinite list of the nonnegative integers.

All three kinds of list arise in everyday programming. Chapter 9 is devoted to exploring the world of infinite lists and their uses. For example, the prelude function `iterate` returns an infinite list:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x:iterate f (f x)
```

In particular, `iterate (+1) 1` is an infinite list of the positive integers, a value we can also write as `[1..]` (see the following section).

As another example,

```
head (filter perfect [1..])
 where perfect n = (n == sum (divisors n))
```

returns the first perfect number, namely 6, even though nobody currently knows whether `filter perfect [1..]` is an infinite or partial list.

Finally, we can define

```
until p f = head . filter p . iterate f
```

The function `until` was used to compute floors in the previous chapter. As this example demonstrates, functions that seem basic in programming are often composed of even simpler functions. A bit like protons and quarks.

4.2 Enumerations

Haskell provides useful notation for enumerating lists of integers. When m and n are integers we can write

`[m..n]`	for the list $[m, m+1, \ldots, n]$
`[m..]`	for the infinite list $[m, m+1, m+2, \ldots]$
`[m,n..p]`	for the list $[m, m+(n-m), m+2(n-m), \ldots, p]$
`[m,n..]`	for the infinite list $[m, m+(n-m), m+2(n-m), \ldots]$

The first two notations crop up frequently in practice, the second two less so. As examples,

```
ghci> [0,2..11]
[0,2,4,6,8,10]
ghci> [1,3..]
[1,3,5,7,9,11 {Interrupted}
```

In the first example the enumeration stops at 10 because 11 isn't even. In the second example we quickly interrupted the evaluation of an infinite list.

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`. We won't elaborate more on this class, except to say that `Char` is also a member:

```
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

4.3 List comprehensions

Haskell provides another useful and very attractive piece of notation, called *list comprehensions*, for constructing lists out of other lists. We illustrate with a few examples:

```
ghci> [x*x | x <- [1..5]]
[1,4,9,16,25]
ghci> [x*x | x <- [1..5], isPrime x]
[4,9,25]
ghci> [(i,j) | i <- [1..5], even i, j <- [i..5]]
[(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
ghci> [x | xs <- [[(3,4)],[(5,4),(3,2)]], (3,x) <- xs]
[4,2]
```

Here is another example. Suppose we wanted to generate all Pythagorean triads in a given range. These are triples of numbers (x,y,z) such that $x^2+y^2=z^2$ and $1 \leq x,y,z \leq n$ for some given n. We can define

```
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | x <- [1..n], y <- [1..n],
                      z <- [1..n], x*x+y*y==z*z]
```

Hence

```
ghci> triads 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),
 (8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

That's probably not what we want: each essentially distinct triad is generated in two different ways. Moreover, the list contains redundant triads consisting of multiples of basic triads.

To improve the definition of `triad` we can restrict x and y so that $x < y$ and x and y are coprime, meaning they have no divisors in common. As mathematicians we know that $2x^2$ cannot be the square of an integer, so the first restriction is valid. The divisors of a number can be computed by

```
divisors x = [d | d <- [2..x-1], x `mod` d == 0]
```

Hence

```
coprime x y = disjoint (divisors x) (divisors y)
```

We will leave the definition of `disjoint` as an exercise.

That means we can define

```
triads n = [(x,y,z) | x <- [1..n], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
```

This definition is better than before, but let us try to make it a little faster, mainly to illustrate an important point. Since $2x^2 < x^2 + y^2 = z^2 \leq n^2$ we see that $x < n/\sqrt{2}$. So $x \leq \lfloor n/\sqrt{2} \rfloor$. That suggests we can write

```
triads n = [(x,y,z) | x <- [1..m], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
           where m = floor (n / sqrt 2)
```

But the expression for `m` is incorrect: `n` is an `Int` and we cannot divide integers. We need an explicit conversion function, and the one to use is `fromIntegral` (not `fromInteger` because `n` is an `Int` not an `Integer`). We need to replace the definition of `m` by `m = floor (fromIntegral n / sqrt 2)`. Once again we have to be careful about what kinds of number we are dealing with and aware of the available conversion functions between them.

List comprehensions can be used to define some common functions on lists. For example,

```
map f xs    = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
concat xss  = [x | xs <- xss, x <- xs]
```

Actually, in Haskell it is the other way around: list comprehensions are translated into equivalent definitions in terms of `map`, and `concat`. The translation rules are:

```
[e |True]         = [e]
[e | q]           = [e | q, True]
[e | b, Q]        = if b then [e | Q] else []
[e | p <- xs, Q] = let ok p = [e | Q]
                       ok _ = []
                   in concat (map ok xs)
```

The definition of `ok` in the fourth rule uses a *don't care* pattern, also called a *wild card*. The `p` in the fourth rule is a pattern, and the definition of `ok` says that the empty list is returned on any argument that doesn't match the pattern `p`.

Another useful rule is

```
[e | Q1, Q2] = concat [[e | Q2] | Q1]
```

4.4 Some basic operations

We can define functions over lists by pattern matching. For example,

```
null :: [a] -> Bool
null []     = True
null (x:xs) = False
```

The patterns `[]` and `x:xs` are disjoint and exhaustive, so we can write the two equations for `null` in either order. The function `null` is strict because Haskell has to know which equation to apply and that requires evaluation of the argument, at least to the extent of discovering whether it is the empty list or not. (A question: why not simply define `null = (==[])`?) We could also have written

```
null [] = True
null _  = False
```

This definition uses a don't care pattern.

Here are two other definitions using pattern matching:

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

There is no equation for the pattern `[]`, so Haskell reports an error if we try to evaluate `head []` or `tail []`.

We can use `[x]` as shorthand for `x:[]` in a pattern:

```
last :: [a] -> a
last [x]      = x
last (x:y:ys) = last (y:ys)
```

The first equation has a pattern that matches a singleton list; the second has a pattern that matches a list that contains at least two elements. The standard prelude definition of `last` is slightly different:

```
last [x]    = x
last (_:xs) = last xs
```

This definition uses a don't care pattern. The two equations have to be written in this order because `x:[]` matches both patterns.

4.5 Concatenation

Here is the definition of `(++)`, the concatenation operation:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

The definition uses pattern matching on the first argument but not on the second. The second equation for `(++)` is very succinct and requires some thought, but once you have got it, you have understood a lot about how lists work in functional programming. Here is a simple evaluation sequence:

```
  [1,2] ++ [3,4,5]
=   {notation}
  (1:(2:[])) ++ (3:(4:(5:[])))
=   {second equation for ++}
  1:((2:[]) ++ (3:(4:(5:[]))))
=   {and again}
  1:(2:([] ++ (3:(4:(5:[])))))
=   {first equation for ++}
  1:(2:(3:(4:(5:[]))))
=   {notation}
  [1,2,3,4,5]
```

As this example suggests, the cost of evaluating `xs++ys` is proportional to the length of `xs`, where

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Note also that

```
undefined ++ [1,2] = undefined
[1,2] ++ undefined = 1:2:undefined
```

We know nothing about the first list, but we do know that the second list begins with 1 followed by 2.

Concatenation is an associative operation. Thus

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

for *all* lists `xs`, `ys` and `zs`. We will see how to prove assertions like these in Chapter 6.

4.6 `concat`, `map` and `filter`

Three very useful list operations that we have met already are `concat`, `map` and `filter`. Here are their definitions using pattern matching:

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x:map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs
```

There is a common theme underlying these definitions that we will identify and exploit in Chapter 6. An alternative definition of `filter` is

```
filter p = concat . map (test p)
test p x = if p x then [x] else []
```

With this definition, `filter p` is implemented by converting each element of the list into a singleton list if it satisfies `p`, and the empty list otherwise. The results are then concatenated.

Two basic facts about `map` are that

```
map id      = id
map (f . g) = map f . map g
```

The first equation says that applying the identity function to each element of a list leaves the list unchanged. The two occurrence of `id` in this law have different types: on the left it is `a -> a` and on the right it is `[a] -> [a]`. The second equation says that applying `g` to every element of a list, and then applying `f` to every element of the result, gives the same list as applying `f . g` to every element. Read from right to left, the equation says that two traversals of a list can be replaced by one, with a corresponding gain in efficiency.

The two facts have a name: they are called the *functor* laws of `map`. The name is borrowed from a branch of mathematics called Category Theory. In fact, Haskell provides a type class `Functor`, whose definition is

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The method `fmap` is expected to satisfy exactly the same laws as `map`. The reason for this type class is that the idea of mapping a function over a list can be generalised to one of mapping a function over an arbitrary data structure, such as trees of various kinds. For example, consider the type

```
data Tree a = Tip a | Fork (Tree a) (Tree a)
```

of binary trees with labels in their tips. Tree-structured data arise in a number of places, for example with the syntax of expressions of various kinds. We can define a mapping function over trees, but rather than calling it `mapTree` we can call it `fmap` by making trees a member of the `Functor` class:

```
instance Functor Tree where
  fmap f (Tip x)    = Tip (f x)
  fmap f (Fork u v) = Fork (fmap f u) (fmap f v)
```

In fact `map` is just a synonym for the instance `fmap` for lists:

```
ghci> fmap (+1) [2,3,4]
[3,4,5]
```

We mention the `Functor` type class here primarily to show that if ever you think some function on lists can be usefully generalised to other kinds of data structure, the chances are good that the designers of Haskell have already spotted it and introduced an appropriate type class. As we will see later on, and especially in Chapter 12, the functor laws of `map` appear in many calculations.

There is another group of laws that involve `map`, all of which have a common theme. Consider the equations

```
f . head       = head . map f
map f . tail   = tail . map f
map f . concat = concat . map (map f)
```

The first equation holds only if `f` is a strict function, but the others hold for arbitrary `f`. If we apply both sides of the equation to the empty list, we get

```
f (head []) = head (map f []) = head []
```

Since the head of an empty list is undefined, we require `f` to be strict to make the equation true.

Each of the laws has a simple interpretation. In each case you can apply the operation (`head`, `tail`, and so on) to a list and then change each element, or you can change each element first and then apply the operation. The common theme lies in the types of the operations involved:

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

The point about the operations is that they do not depend in any way on the nature of the list elements; they are simply functions that shuffle, discard or extract elements from lists. That is why they have polymorphic types. And functions with polymorphic types all satisfy some law that says you can change values before or after applying the function. In mathematics such functions are called *natural transformations* and the associated laws, *naturality* laws.

As another example, since `reverse :: [a] -> [a]` we would expect that

```
map f . reverse =  reverse . map f
```

Indeed this is the case. Of course, this naturality law still has to be proved.

Another law is

```
concat . map concat = concat . concat
```

The two sides assert that two ways of concatenating a list of lists of lists (either do the inner concatenations first, or do the outer concatenations first) give the same result.

Finally, here is just one property of `filter`:

```
filter p . map f = map f . filter (p . f)
```

We can prove this law by simple equational reasoning:

```
  filter p . map f
=   {second definition of filter}
  concat . map (test p) . map f
=   {functor property of map}
  concat . map (test p . f)
=   {since test p . f = map f . test (p . f)}
  concat . map (map f . test (p . f))
=   {functor property of map}
  concat . map (map f) . map (test (p . f))
=   {naturality of concat}
  map f . concat . map (test (p . f))
=   {second definition of filter}
  map f . filter (p . f)
```

Laws like those above are not just of academic interest, but are deployed in finding new and better ways of expressing definitions. That's why functional programming is the best thing since sliced bread.

4.7 zip and zipWith

Finally, to complete a simple toolbox of useful operations, we consider the functions zip and zipWith. The definitions in the standard prelude are:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _      _      = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

A caring programmer (one who doesn't like 'don't care' patterns) would have written

```
zip [] ys         = []
zip (x:xs) []     = []
```

```
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Both definitions use pattern matching on both arguments. You have to know that pattern matching is applied from top to bottom and from left to right. Thus

```
zip [] undefined = []
zip undefined [] = undefined
```

The definition of `zip` can be given another way:

```
zip = zipWith (,)
```

The operation `(,)` is a constructor for pairs: `(,) a b = (a,b)`.

Here is one example of the use of `zipWith`. Suppose we want to determine whether a list is in nondecreasing order. A direct definition would have:

```
nondec :: (Ord a) => [a] -> Bool
nondec []       = True
nondec [x]      = True
nondec (x:y:xs) = (x <= y) && nondec (y:xs)
```

But another, equivalent and shorter definition is

```
nondec xs = and (zipWith (<=) xs (tail xs))
```

The function `and` is yet another useful function in the standard prelude. It takes a list of booleans and returns `True` if all the elements are `True`, and `False` otherwise:

```
and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

One final example. Consider the task of building a function `position` that takes a value `x` and a finite list `xs` and returns the first position in `xs` (counting positions from 0) at which `x` occurs. If `x` does not occur in the list, then -1 is returned. We can define

```
position :: (Eq a) => a -> [a] -> Int
position x xs
  = head ([j | (j,y) <- zip [0..] xs, y==x] ++ [-1])
```

The expression `zip [0..] xs` pairs each element of `xs` with its position in `xs`. Although the first argument of `zip` is an infinite list, the result is a finite list whenever `xs` is. Observe that the problem is solved by first computing the list of *all* positions at which `x` is found, and then taking the first element. Under lazy evaluation it is not necessary to construct the value of every element of the list in order

to calculate the head of the list, so there is no great loss of efficiency in solving the problem this way. And there is a great deal of simplicity in defining one search result in terms of all search results.

4.8 Common words, completed

Let's now return to Section 1.3 and complete the definition of `commonWords`. Recall that we finished with

```
commonWords :: Int -> [Char] -> [Char]
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

The only functions we have still to give definitions for are

```
showRun   countRuns   sortRuns   sortWords
```

All the others, including `words`, are provided in the standard Haskell libraries.

The first one is easy:

```
showRun :: (Int,Word) -> [Char]
showRun (n,w) = w ++ ": " ++ show n ++ "\n"
```

The second one can be defined by

```
countRuns :: [Word] -> [(Int,Word)]
countRuns []     = []
countRuns (w:ws) = (1+length us,w):countRuns vs
                   where (us,vs) = span (==w) ws
```

The prelude function `span p` splits a list into two, the first being the longest prefix of the list all of whose elements satisfy the test `p`, and the second being the suffix that remains. Here is the definition:

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span p []     = ([],[])
span p (x:xs) = if p x then (x:ys,zs)
                else ([],x:xs)
                where (ys,zs) = span p xs
```

That leaves `sortRuns` and `sortWords`. We can import the function `sort` from `Data.List` by the command

```
import Data.List (sort)
```

Since `sort :: (Ord a) => [a] -> [a]` we can then define

```
sortWords :: [Word] -> [Word]
sortWords = sort

sortRuns :: [(Int,Word)] -> [(Int,Word)]
sortRuns = reverse . sort
```

To understand the second definition you have to know that Haskell automatically defines the comparison operation `(<=)` on pairs by

```
(x1,y1) <= (x2,y2) = (x1 < x2) || (x1 == x2 && y1 <= y2)
```

You also have to know that `sort` sorts into ascending order. Since we want the codes in descending order of count, we just sort into ascending order and reverse the result. That, by the way, is why we defined frequency counts by having the count before the word rather than afterwards.

Instead of relying on the library function for sorting, let us end by programming a sorting function ourselves. One good way to sort is to use a *divide and conquer* strategy: if the list has length at most one then it is already sorted; otherwise we can divide the list into two equal halves, sort each half by using the sorting algorithm recursively, and then merge the two sorted halves together. That leads to

```
sort :: (Ord a) => [a] -> [a]
sort []  = []
sort [x] = [x]
sort xs  = merge (sort ys) (sort zs)
           where (ys,zs) = halve xs

halve xs = (take n xs, drop n xs)
           where n = length xs `div` 2
```

That leaves us with the definition of `merge`, which merges two sorted lists together into one sorted list:

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

In fact, many Haskell programmers wouldn't write the last clause of `merge` in quite this way. Instead they would write

```
merge xs'@(x:xs) ys'@(y:ys)
  | x <= y    = x:merge xs ys'
  | otherwise = y:merge xs' ys
```

This definition uses an *as-pattern*. You can see the point: rather than deconstructing a list and then reconstructing it again (a cheap but not free operation), it is better to reuse the value that we matched with. True, but it does obscure a simple mathematical equation, and we will use such patterns only very sparingly in this book.

Both `sort` and `merge` are defined recursively and it is worthwhile pointing out why the two recursions terminate. In the case of `merge` you have to see that one or other of the two arguments of `merge` decreases in size at each recursive call. Hence one of the base cases will eventually be reached. In the case of `sort` the critical observation is that if `xs` has length at least two, then both `ys` and `zs` have length strictly less than `xs`, and the same argument applies. But see what happens if we had omitted the clause `sort [x] = [x]`. Since $1 \operatorname{div} 2 = 0$ we would have,

```
sort [x] = merge (sort []) (sort [x])
```

That means evaluation of `sort [x]` requires evaluation of `sort [x]`, and the whole definition of `sort` spins off into an infinite loop for nonempty arguments. Checking that you have all the necessary base cases is one of the most important parts of constructing a recursive function.

4.9 Exercises

Exercise A

Which of the following equations are true for all `xs` and which are false?

```
[]:xs = xs
[]:xs = [[],xs]
xs:[] = xs
xs:[] = [xs]
xs:xs = [xs,xs]
[[]] ++ xs = xs
[[]] ++ xs = [[],xs]
[[]] ++ [xs] = [[],xs]
```

```
[xs] ++ [] = [xs]
```

By the way, why didn't we define `null = (==[])`?

Exercise B

You want to produce an infinite list of all distinct pairs (x,y) of natural numbers. It doesn't matter in which order the pairs are enumerated, as long as they all are there. Say whether or not the definition

```
allPairs = [(x,y) | x <- [0..], y <- [0..]]
```

does the job. If you think it doesn't, can you give a version that does?

Exercise C

Give a definition of the function

```
disjoint :: (Ord a) => [a] -> [a] -> Bool
```

that takes two lists in ascending order, and determines whether or not they have an element in common.

Exercise D

Under what conditions do the following two list comprehensions deliver the same result?

```
[e | x <- xs, p x, y <- ys]
[e | x <- xs, y <- ys, p x]
```

Compare the costs of evaluating the two expressions.

Exercise E

When the great Indian mathematician Srinivasan Ramanujan was ill in a London hospital, he was visited by the English mathematician G.H. Hardy. Trying to find a subject of conversation, Hardy remarked that he had arrived in a taxi with the number 1729, a rather boring number it seemed to him. Not at all, Ramanujan instantly replied, it is the first number that can be expressed as two cubes in essentially different ways: $1^3+12^3 = 9^3+10^3 = 1729$. Write a program to find the second such number.

In fact, define a function that returns a list of all essentially different quadruples (a,b,c,d) in the range $0 < a,b,c,d \leq n$ such that $a^3+b^3 = c^3+d^3$. I suggest using a list comprehension, but only after thinking carefully about what it means to say

two quadruples are essentially different. After all, $a^3+b^3=c^3+d^3$ can be written in eight different ways.

Exercise F

The dual view of lists is to construct them by adding elements to the end of the list:

```
data List a = Nil | Snoc (List a) a
```

`Snoc` is, of course, `Cons` backwards. With this view of lists $[1,2,3]$ would be represented by

```
Snoc (Snoc (Snoc Nil 1) 2) 3
```

Exactly the same information is provided by the two views but it is organised differently. Give the definitions of `head` and `last` for the snoc-view of lists, and define two functions

```
toList :: [a] -> List a
fromList :: List a -> [a]
```

for converting efficiently from one view of lists to the other. (Hint: `reverse` is efficient, taking linear time to reverse a list.)

Exercise G

How much space is required to evaluate `length xs`? Consider the following alternative definition of `length`:

```
length :: [a] -> Int
length xs = loop (0,xs)
  where loop (n,[])   = n
        loop (n,x:xs) = loop (n+1,xs)
```

Does the space requirement change? Does it change if we switched to eager evaluation? These questions are taken up in much more detail in Chapter 7.

Exercise H

The prelude function `take n` takes the first `n` elements of a list, while `drop n` drops the first `n` elements. Give recursive definitions for these functions. What are the values of

```
take 0 undefined     take undefined []
```

according to your definition? A more tricky question: can you find a definition in which both the above expressions have the value `[]`? If not, why not?

Which of the following equations are valid for all integers *m* and *n*? You don't have to justify your answers, just try to understand what they claim to say.

```
take n xs ++ drop n xs  =  xs
take m . drop n  =  drop n . take (m+n)
take m . take n  =  take (m `min` n)
drop m . drop n  =  drop (m+n)
```

The standard prelude function `splitAt n` can be defined by

```
splitAt n xs = (take n xs,drop n xs)
```

Though clear, the above definition is maybe a little inefficient as it involves processing `xs` twice. Give a definition of `splitAt` that traverses the list only once.

Exercise I

Which of the following statements about the equation

```
map (f . g) xs = map f (map g xs)
```

do you agree with, and which do you disagree with (again, no justification is required)?

1. It's not true for all `xs`; it depends on whether `xs` is a finite list or not.
2. It's not true for all `f` and `g`; it depends on whether `f` and `g` are strict functions or not.
3. It's true for all lists `xs`, finite, partial or infinite, and for all `f` and `g` of the appropriate type. In fact `map (f . g) = map f . map g` is a much neater alternative.
4. It looks true, but it has to be proved so from the definition of `map` and the definition of functional composition.
5. Used right-to-left, it expresses a program optimisation: two traversals of a list are replaced by one.
6. It's not an optimisation under lazy evaluation because `map g xs` is not computed in its entirety before evaluation of `map f` on the result begins.
7. Whether or not it is computed in pieces or as a whole, the right-hand side does produce an intermediate list, while the left-hand side doesn't. It is a rule for optimising a program even under lazy evaluation.

Exercise J

Here are some equations; at least one of them is false. Which are the true ones, and which are false? Once again, you do not have to provide any justification for your answers, the aim is just to look at some equations and appreciate what they are saying.

```
map f . take n    = take n . map f
map f . reverse   = reverse . map f
map f . sort      = sort . map f
map f . filter p  = map fst . filter snd . map (fork (f,p))
filter (p . g)    = map (invertg) . filter p . map g
reverse . concat  = concat . reverse . map reverse
filter p . concat = concat . map (filter p)
```

In the fifth equation assume `invertg` satisfies `invertg . g = id`. The function `fork` in the fourth equation is defined by

```
fork :: (a -> b,a -> c) -> a -> (b,c)
fork (f,g) x = (f x, g x)
```

Exercise K

Define `unzip` and `cross` by

```
unzip = fork (map fst, map snd)
cross (f,g) = fork (f . fst, g . snd)
```

What are the types of these functions?

Prove by simple equational reasoning that

```
cross (map f, map g) . unzip = unzip . map (cross (f,g))
```

You can use the functor laws of `map` and the following rules:

```
cross (f,g) . fork (h,k) = fork (f . h,g . k)
fork (f,g) . h           = fork (f . h,g . h)
fst . cross (f,g)        = f . fst
snd . cross (f,g)        = g . snd
```

Exercise L

Continuing from the previous exercise, prove that

```
cross (f,g) . cross (h,k) = cross (f . h,g . k)
```

We also have `cross (id,id) = id` (Why?). So it looks like `cross` has functor-like properties, except that it takes a pair of functions. Yes, it's a *bifunctor*. That suggests a generalisation:

```
class Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
```

The arguments to `bimap` are given one by one rather than paired. Express `cross` in terms of `bimap` for the instance `Pair` of `Bifunctor`, where

```
type Pair a b = (a,b)
```

Now consider the data type

```
data Either a b = Left a | Right b
```

Construct the instance `Either` of `Bifunctor`.

4.10 Answers

Answer to Exercise A

Only the following three equations are true:

```
xs:[] = [xs]
[[]] ++ [xs] = [[],xs]
[xs] ++ [] = [xs]
```

If we defined `null` by `null = (==[])`, then its type would have to be the more restrictive

```
null :: (Eq a) => [a] -> Bool
```

That means you can only use an equality test on lists if the list elements can be compared for equality. Of course, the empty list contains no elements, so `(==)` is not needed.

Answer to Exercise B

No, `allPairs` produces the infinite list

```
allPairs = [(0,y) | y <- [0..]]
```

One alternative, which lists the pairs in ascending order of their sum, is

```
allPairs = [(x,d-x) | d <- [0..], x <- [0..d]]
```

Answer to Exercise C

The definition is

```
disjoint xs [] = True
disjoint [] ys = True
disjoint xs'@(x:xs) ys'@(y:ys)
  | x < y  = disjoint xs ys'
  | x == y = False
  | x > y  = disjoint xs' ys
```

We used an as-pattern, just to be clever.

Answer to Exercise D

They deliver the same result only if `ys` is a finite list:

```
ghci> [1 | x <- [1,3], even x, y <- undefined]
[]
ghci> [1 | x <- [1,3], y <- undefined, even x]
*** Exception: Prelude.undefined
ghci> [1 | x <- [1,3], even x, y <- [1..]]
[]
Prelude> [1 | x <- [1,3], y <- [1..], even x]
{Interrupted}
```

When they do deliver the same result, the former is more efficient.

Answer to Exercise E

One way of generating essentially different quadruples is to restrict the quadruple (a,b,c,d) to values satisfying $a \leq b$ and $c \leq d$ and $a < c$. Hence

```
quads n = [(a,b,c,d) | a <- [1..n],  b <- [a..n],
                       c <- [a+1..n],d <- [c..n],
                       a^3 + b^3 == c^3 + d^3]
```

The second such number is $4104 = 2^3 + 16^3 = 9^3 + 15^3$.

Answer to Exercise F

```
head :: List a -> a
head (Snoc Nil x) = x
head (Snoc xs x)  = head xs
```

```
last :: List a -> a
last (Snoc xs x) = x

toList :: [a] -> List a
toList = convert . reverse
   where convert []     = Nil
         convert (x:xs) = Snoc (convert xs) x
fromList :: List a -> [a]
fromList = reverse . convert
   where convert Nil         = []
         convert (Snoc xs x) = x:convert xs
```

Answer to Exercise G

It requires a linear amount of space since the expression

```
1 + (1 + (1 + ... (1 + 0)))
```

is built up in memory. The space requirement for the second definition of length does not change under lazy evaluation since the expression

```
loop ((((0 + 1) + 1) + 1 ... +1),[])
```

is built up in memory. But under eager evaluation the length of a list can be computed using constant extra space.

Answer to Exercise H

```
take, drop :: Int -> [a] -> [a]
take n []     = []
take n (x:xs) = if n==0 then [] else x:take (n-1) xs

drop n []     = []
drop n (x:xs) = if n==0 then x:xs else drop (n-1) xs
```

With this definition of `take` we have

```
take undefined [] = []   take 0 undefined = undefined
```

With the alternative

```
take n xs | n==0      = []
          | null xs   = []
          | otherwise = head xs: take (n-1) (tail xs)
```

we have

```
take undefined [] = undefined   take 0 undefined = []
```

The answer to the tricky question is: no. Either argument `n` or argument `xs` has to be examined and, whichever happens first, $\perp$ is the result if $\perp$ is the value of that argument.

All four equations are valid for all lists *xs* and for all $m, n \neq \perp$, under either definition.

The function `splitAt n` can be defined by

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n [] = ([],[])
splitAt n (x:xs) = if n==0 then ([],x:xs) else (x:ys,zs)
                   where (ys,zs) = splitAt (n-1) xs
```

Answer to Exercise I

I would agree with (3), (4), (5) and (7).

Answer to Exercise J

The only false equation is `map f . sort = sort . map f` which is true only if f is order-preserving, i.e. $x \leq y \equiv f\ x \leq f\ y$.

Answer to Exercise K

```
unzip :: [(a,b)] -> ([a],[b])
cross :: (a -> b, c -> d) -> (a,c) -> (b,d)
```

The calculation is

```
    cross (map f, map g) . unzip
=     {definition of unzip}
    cross (map f, map g) . fork (map fst, map snd)
=     {law of cross and fork}
    fork (map f . map fst, map g . map snd)
=     {law of map}
    fork (map (f . fst), map (g . snd))
```

We seem to be stuck, as no law applies. Try the right-hand side:

```
  unzip . map (cross (f,g))
=   {definition of unzip}
  fork (map fst, map snd) . map (cross (f,g))
=   {law of fork}
  fork (map fst . map (cross (f,g)),
        map snd . map (cross (f,g)))
=   {law of map}
  fork (map (fst . cross (f,g)),
        map (snd . cross (f,g)))
=   {laws of fst and snd}
  fork (map (f . fst), map (g . snd))
```

Phew. Both sides have reduced to the same expression. That is often the way with calculations: one side doesn't always lead easily to the other, but both sides reduce to the same result.

The calculations we have seen so far have all been carried out at the function level. Such a style of definition and proof is called *point-free* (and also *pointless* by some jokers). Point-free proofs are what the automatic calculator of Chapter 12 produces. The point-free style is very slick, but it does necessitate the use of various *plumbing combinators*, such as `fork` and `cross`, to pass arguments to functions. Plumbing combinators push values around, duplicate them and even eliminate them. As an example of the last kind,

```
const :: a -> b -> a
const x y = x
```

This little combinator is in the standard prelude and can be quite useful on occasion.

Two more plumbing combinators, also defined in the standard prelude, are `curry` and `uncurry`:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

A *curried* function is a function that takes its arguments one at a time, while a non-curried function takes a single, tupled argument. The key advantage of curried

functions is that they can be *partially applied*. For instance, `take n` is a perfectly valid function in its own right, and so is `map f`. That is why we have used curried functions from the start.

By the way, curried functions are named after Haskell B. Curry, an American logician. And, yes, that is where Haskell got its name.

Answer to Exercise L

```
  cross (f,g) . cross (h,k)
=   {definition of cross}
  cross (f,g) . fork (h . fst, k . snd)
=   {law of cross and fork}
  fork (f . h . fst,g . k . snd)
=   {definition of cross}
  cross (f . h, g . k)
```

We have `cross = uncurry bimap`, where `uncurry` was defined in the previous answer.

Here is the instance of `Either`:

```
instance Bifunctor Either where
  bimap f g (Left x)  = Left (f x)
  bimap f g (Right y) = Right (g y)
```

4.11 Chapter notes

Most of the functions introduced in this chapter can be found in the Haskell standard prelude. Functors, bifunctors, and natural transformations are explained in books about Category Theory. Two such are *Basic Category Theory for Computer Scientists* (MIT Press, 1991) by Benjamin Pierce, and *The Algebra of Programming* (Prentice Hall, 1997) by Richard Bird and Oege de Moor.

Also on the subject of laws, read Phil Wadler's influential article *Theorems for free!* which can be found at

```
homepages.inf.ed.ac.uk/wadler/papers/free/
```