Applications of these operators include computing the minimum or the maximum of a list:

```
maximum xs = foldl1 max xs
minimum xs = foldl1 min xs
```

2.5.4 Filters

A common requirement is to traverse a list and discover values which satisfy a certain predicate. *Filtering* is the process of forming a new list by selecting elements of the original list which satisfy a predicate. The predicate is a function which returns either `True` or `False`, and is passed as an argument to the `filter` function.

For example (`even` is predefined in the Standard Prelude):

```
filter even [45, 17, 32, 3, 8, 9, 76]  ⇒  [32, 8, 76]
```

We define `filter` as:

```
filter _ []                  = []
filter p (x:xs) | p x        = x : filter p xs
                | otherwise  = filter p xs
```

Alternatively, we may only be interested in whether any value in the list *does* satisfy the predicate:

```
any _ []                  = False
any p (x:xs) | p x        = True
             | otherwise  = any p xs
```

We can use `any` to redefine the function `elem` (see Figure 2.1) which checks whether an element is a member of a list:

```
elem n xs = any (\x -> x == n) xs
```

There are two other useful functions: the function `takeWhile` has the same type as `filter` and collects all elements of the list but stops at the first element that does not satisfy the predicate; `dropWhile` works similarly but it drops all the elements until one does not satisfy the predicate:

```
takeWhile _ []                    = []
takeWhile p (x:xs)  | p x         = x: (takeWhile p xs)
                    | otherwise   = []

dropWhile _ []                    = []
dropWhile p xs@(x:xs') | p x      = dropWhile p xs'
                       | otherwise = xs
```


2.5.5 Functions in data structures

Functions can be placed into data structures such as lists and user-defined structures. For example:

```
double x          = x * 2
square x          = x * x
bump   x          = x + 1

applyall [] x     = x
applyall (f:fs) x = f (applyall fs x)

y = applyall [bump, square, double] 3
```

The function `applyall` applies a list of functions (which must all have the same type signature) to an initial value. In the above example, `y` would become equal to:

```
bump (square (double 3))
```

which equals $37 = (3 \times 2)^2 + 1$.

2.6 Algebraic types and polymorphism

In Section 2.4.1, we discussed the structure of a list in terms of applications of the constructor `(:)`. This section describes constructors in general and how their types can be expressed. It also introduces polymorphism as a concept which allows one function to operate on different types.

2.6.1 User-defined types

Haskell allows us to define our own types, using *constructors*. A constructor is just like a function, expecting some arguments and delivering a value of the type it defines.

For example, suppose we want a data type to define values which store coordinates in the cartesian plane. We can give a new name to that type, say `CoordType`. We then define a constructor function, call it `Coord`, with two arguments, the x and y components, that returns a value of `CoordType`.

In Haskell we introduce such a type with the following declaration:

```
data CoordType = Coord Float Float deriving Show
```

`deriving Show` at the end of the definition is necessary for user-defined types to make sure that values of these types can be printed in a standard form, as will be explained in Section 2.8.5.

If we want to refer to an actual coordinate (that is, a value of this type) in a program, we write it as an application of the `Coord` constructor. For example, the coordinate which has the x component as `14.0` and the y component as `2.0` can be written as: