```
? minimum [100,99..1]
1
(809 reductions, 1322 cells)
?
```

(The expression [100,99..1] denotes the list of integers from 1 to 100 arranged in decreasing order, as described in section 10.1).

A rather different approach involves sorting the elements of the list into increasing order (using the function sort defined in the standard prelude) and then take the element at the head of the resulting list (using the standard function head). Of course, sorting the list in its entirety is likely to require significantly more work than the previous approach:

```
? sort [100,99..1]
[1, 2, 3, 4, 5, 6, 7, 8, ... etc ..., 99, 100]
(10712 reductions, 21519 cells)
?
```

However, thanks to lazy-evaluation, calculating just the first element of the sorted list actually requires less work in this particular case than the first solution using minimum:

```
? head (sort [100,99..1])
1
(713 reductions, 1227 cells)
?
```

Incidentally, it is probably worth pointing out that this example depends rather heavily on the particular algorithm used to sort a list of elements. The results are rather different if we compare the same two approaches used to calculate the maximum value in the list:

```
? maximum [100,99..1]
100
(812 reductions, 1225 cells)
? last (sort [100,99..1])
100
(10612 reductions, 20732 cells)
?
```

This difference is caused by the fact that each element in the list produced by sort is only known once the values of all of the preceding elements are also known. Thus the complete list must be sorted in order to obtain the last element.

9.7 Infinite data structures

One particular benefit of lazy evaluation is that it makes it possible for functions in Gofer to manipulate 'infinite' data structures. Obviously we cannot hope either to construct or store an infinite object in its entirety – the advantage of lazy evaluation is that it allows us to construct infinite objects piece by piece as necessary (and to reuse the storage space used by parts of the object when they are no longer required).

As a simple example, consider the following function which can be used to produce infinite lists of integer values:

```
countFrom n = n : countFrom (n+1)
```

If we evaluate the expression `countFrom 1`, Gofer just prints the list of integer values beginning with 1 until it is interrupted. Once each element in the list has been printed, the storage used to hold that element can be reused to hold later elements in the list. Evaluating this expression is equivalent to using an 'infinite' loop to print the list of integers in an imperative programming language:

```
? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,^C{Interrupted!}
(53 reductions, 160 cells)
?
```

For practical applications, we are usually only interested in using a finite portion of an infinite data structure (just as loops in an imperative programming language are usually terminated after finitely many iterations). For example, using `countFrom` together with the function `take` defined in the standard prelude, we can repeat the calculation from section 4 to find the sum of the integers 1 to 10:

```
? sum (take 10 (countFrom 1))
55
(62 reductions, 119 cells)
?
```

The expression `take n xs` evaluates to a list containing the first `n` elements of the list `xs` (or to `xs` itself if the list contains fewer than n elements). Thus `countFrom 1` generates the infinite list of integers, `take 10` ensures that only the first ten elements are calculated, and `sum` calculates the sum of those integers as before.

A particular advantage of using infinite data structures is that it enables us to describe an object without being tied to one particular application of that object. Consider the following definition for the infinite list of powers of two $[1, 2, 4, 8, \ldots]$:

```
powersOfTwo = 1 : map double powersOfTwo
              where double n = 2*n
```

This list be used in a variety of ways; using the operator `(!!)` defined in the standard prelude (`xs!!n` evaluates to the `n`th element of the list `xs`), we can define a function to find the nth power of 2 for any given integer n:

```
twoToThe n = powersOfTwo !! n
```

Alternatively, we can use the list `powersOfTwo` to define a function mapping lists of bits (represented by integers 0 and 1) to the corresponding decimal number: simply reverse the order of the digits, multiply each by the corresponding power of two and calculate the sum. Using functions from the standard prelude, this translates directly into the definition:

```
binToDec ds = sum (zipWith (*) (reverse ds) powersOfTwo)
```

For example:

```
? twoToThe 12
4096
(15 reductions, 21 cells)
? binToDec [1,0,1,1,0]
22
(40 reductions, 85 cells)
?
```

9.8 Polymorphism

Given the definition of `product` in section 9.5, it is easy to see that product takes a single argument which is a list of integers and returns a single integer value – the product of the elements of the list. In other words, `product` has type `[Int] -> Int`. On the other hand, it is not immediately clear what the type of the function `map` should be. Clearly the first argument of `map` must be a function and both the second argument and the result are lists, so that the type of `map` must be of the form:

$$\underbrace{(a \to b)}_{\text{type of argument } \texttt{f}} \quad \to \quad \underbrace{[c]}_{\text{type of argument } \texttt{xs}} \quad \to \quad \underbrace{[d]}_{\text{type of result } \texttt{map f xs}}$$

But what can be said about the types a, b, c and d? One possibility would be to choose $a = b = c = d =$ `Int` which would be acceptable for expressions such as `map fact [1,2,3,4]`, but this would not be suitable in an expression such as `map chr [65,75,32]` because the `chr` function does not have type `Int -> Int`.

Notice however that the argument type of `f` must be the same as the type of elements in the second argument (i.e. $a = c$) since `f` is applied to each element in that list. Similarly, the result type of `f` must be the same as the type of elements in the result list (i.e. $b = d$) since each element in this list is obtained as a result of applying the function `f` to some value. It is therefore reasonable to treat the `map` function as having any type of the form:

$$(a \to b) \to [a] \to [b]$$

The letters a and b used in this type expression represent arbitrary types and are called type variables. An object whose type includes one or more type variables can be thought of as having many different types and is often described as having a 'polymorphic type' (literally: its type has 'many shapes').

The ability to define and use polymorphic functions in Gofer turns out to be very useful. Here are the types of some of the other polymorphic functions which have been used in previous examples which illustrate this point:

```
length :: [a] -> Int
(++)   :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
```

Thus we can use precisely the same `length` function to determine both the length of a list of integers as well as finding the length of a string:

```
? length [1..10]
10
(98 reductions, 138 cells)
? length "Hello"
5
(22 reductions, 36 cells)
?
```

9.9 Higher-order functions

In Gofer, function values are treated in much the same way as any other kind of value; in particular, they can be used both as arguments to, and results of other functions.

Functions which manipulate other functions in this way are often described as 'higher-order functions'. Consider the following example, taken from the standard prelude:

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

As indicated by the type declaration, we think of the `(.)` operator as a function taking two function arguments and returning another function value as its result. If `f` and `g` are functions of the appropriate types, then `(f . g)` is a function called the composition of `f` with `g`. Applying `(f . g)` to a value is equivalent to applying `g` to that value, and then applying `f` to the result (as described, far more eloquently, by the second line of the declaration above!).

Many problems can often be described very elegantly as a composition of other functions. Consider the problem of calculating the total number of characters used in a list of strings. A simple recursive function provides one solution:

```
countChars []     = 0
countChars (w:ws) = length w + countChars ws

? countChars ["super","cali","fragi","listic"]
20
(96 reductions, 152 cells)
?
```

An alternative approach is to notice that we can calculate the total number of characters by first combining all of the words in the argument list into a single word (using concat) and then finding the length of that word:

```
? (length . concat) ["super","cali","fragi","listic"]
20
(113 reductions, 211 cells)
?
```

Another solution is to first find the length of each word in the list (using the `map` function to apply `length` to each word) and then calculate the sum of these individual lengths:

```
? (sum . map length) ["super","cali","fragi","listic"]
20
(105 reductions, 172 cells)
?
```

9.10 Variable declarations

A variable declaration is a special form of function definition, almost always consisting of a single equation of the form:

var = *rhs*

(i.e. a function declaration of arity 0). Whereas the values defined by function declarations of arity > 0 are guaranteed to be functions, the values defined by variable declarations may or may not be functions: