Lists are fundamentally important in Haskell. Let’s start by trying to define a list, as we would in GAP:

```
Hugs> x := [1,2]
```

We get the error

ERROR - Undefined data constructor ":="

OK, let’s try

```
Hugs> x = [1,2]
```

ERROR - Syntax error in input (unexpected ‘=’)

Well, how do we make an assignment in Haskell? The answer is that Haskell doesn’t really do assignments. We can define something called \textit{x} that will equal the list \([1, 2]\) by reading in the following lines from a file:

```
x :: [Int]
x = [1,2]
```

The first line says that \textit{x} is an object of type a list of objects of type \texttt{Int} (integers), and the second one says that \textit{x} will equal the list \([1, 2]\). If only the first line is read in, then Haskell gives an error since no “binding”, or definition, of \textit{x} was found. The HUGS interpreter will not let you type these in at the prompt— it doesn’t trust you to give a good binding, so it gives you an error as soon as you type in \texttt{x :: [Int]}.

This may seem to be an “assignment,” but Haskell will never let you redefine the value of \textit{x}, which will remain equal to \([1, 2]\) for the rest of the session. The lack of true assignment might seem to be a very restrictive limitation, but in return we avoid all the many difficult-to-avoid and difficult-to-correct errors that arise from changing values of variables.

Still working at the interpreter, let’s explore some of the basics of lists. The expression

```
Hugs> [1,2]
```

evaluates simply to \([1, 2]\). Haskell has some of the same list notations as GAP. Try

```
Hugs> [1..10]
```

It also understands

```
Hugs> [(-15)..(-10)]
```

but the parentheses are necessary to keep it from getting confused.

Lists in Haskell differ from lists in GAP in a very important way. Haskell lists must be \textit{homogeneous}, that is, every element in them must have the same type. Trying to evaluate

```
Hugs> [(1,2),5]
```

gives an error, as does
Hugs> [(1,2),(3,4,5)]
but
Hugs> [(1,2),(3,5)]
is no problem.
Notice that
Hugs> [[1,2],[3,4,5]]
is OK. Unlike tuples, lists of varying lengths are objects of the same type, provided that their entries are the same type. Thus, the type of [1,2] is [Int], and the type of [[1,2],[3,4,5]] is [[Int]]. Note that these are different, so Haskell does not accept a construction such as [[1,2],[[3,4]]] as a list.
It is true that [1,2] has type [Int], but something odd happens when we check the type of [1,2]:
Hugs> :t [1,2]
Expecting
[1,2] :: [Int]
we instead get
[1,2] :: Num a => [a]
This will make more sense later. Notice that Haskell does accept
Hugs> [1, 2, 3.5]
and gives its type to be
[1, 2, 3.5] :: Fractional a => [a]
This involves a more advanced concept in Haskell, called “unification.” Roughly speaking, in some situations Haskell figures out the most general common type of a collection of objects. For [1,2], it determines the class containing 1 and 2 to be the “numerical” and for [1, 2, 3.5] it determines the common class to be “fractional.”
Lists are usually built up by adding elements to the front of the list using the cons operator, for list constructor. The cons operator is denoted by colon, so
Hugs> 5:[1,2]
Hugs> 5:[ ]
Hugs> [1,2]:[[1,2],[3,4,5]]
Hugs> 5:4:3:2:1:[ ]
Actually, this last expression is what Haskell actually “sees”; when you write [5,4,3,2,1], the Haskell interpreter immediately parses it to 5:4:3:2:1:[ ] for further evaluation.
One of the important built-in types is the character type. Characters are indicated by
single quotes, as in

Hugs> 'a'
'a'

Double quotes indicate a character “string”, which is a list of characters. To test this, evaluate

Hugs> 'a':'b':[ ]

Lists are so fundamental in Haskell that many of the common list functions are in the Prelude. Here are a few of the most common:

Hugs> concat :: [[a]] -> [a]
is the concatenate function. It combines a list of lists into one single list:

Hugs> concat [[1,2,3], [4], [], [4,5,6]]
[1, 2, 3, 4, 4, 5, 6]

Hugs> elem :: Eq a => a -> [a] -> Bool
is the is element of function. It tell whether an element of type a is a member of a list of elements of type a, where the type a must have a concept of equality.

Hugs> elem 5 [1..10]
True

The head function

Hugs> head :: [a] -> a
simply returns the first element of the list.

Hugs> head "abcdef"
'a'

The list must be nonempty,

Hugs> head []
produces an error.

There is also a last function, but a more natural companion function to head is

Hugs> tail :: [a] -> [a]
which returns the list with its first element removed. Again, the list must be nonempty.

Lists can be added using the concatenation operator ++, provided of course that they are of the same type. Thus we have

Hugs> "abc" ++ 'a':"bc"
"abcabc"

Note that the precedence of operations is that cons precedes concatenation. In general, cons has one of the highest precedences of any of the Haskell operators.
Can Haskell understand this input?

Hugs> "The answer is " ++ (5 * 5)

Most computer languages such as Java or C will make the type conversion needed to interpret this line, but Haskell is very strongly typed and gives an error to this input. There is a function for converting numerical (and many other) expressions into strings, called the show function. If you enter

Hugs> "The answer is " ++ show (5 * 5)

the show function converts 25 to a string, which is then concatenated with the first string.