Working with lists

In this section we learn some very basic list operations.

Reading

- Sections 2.1-2.5
- Practice exercises (2.7): 2, 3, 4, 5 (definitely do 4, 5)

Lists in Haskell

Lists of elements are one of the most primitive and most useful value types in Haskell. A list is simply a *sequence of zero or more elements of the same type*. These are defining characteristics of a list:

- All elements in a list must be of the same type (can't mix numbers and strings).
- A list can contain any number of elements.
- The elements in a list are accessed in a particular order.

We can create a list in two ways:

- By enclosing the desired elements in square brackets, and separated by commas: `[4, 6, 1, 2]`. A special notation `['a'..'g']`, called *enumeration*, allows us to form a list of all values from a specific value to another.
- By appending an element to the front of an existing list: `x:xs`. Here `x` is the new element, and `xs` is the list of existing elements. For instance the list `[1, 2]` can be written as `1:[2]` or also as `1:2:[]`. In this way lists are a little like linked lists in other languages.

There are many built-in functions that work on lists. They are all part of what is known as "Standard Prelude", and most can be seen in appendix B.8 from the book. You can also access the online documentation[1], though that takes a bit getting used to.

> Returns the first element of a non-empty list.
>
> Returns all but the first element of a non-empty list.
>
> Given an integer `n` and a list, returns the first `n` elements from the list.

[1] https://hackage.haskell.org/package/base-4.10.0.0/docs/Prelude.html#g:13

Given an integer `n` and a list, returns the remaining of the list after the first `n` elements are removed.

Returns the length of a list.

Returns the sum of the elements in the list, assuming they can be added.

Returns the product of the elements in the list, assuming they can be multiplied.

Appends to lists.

Reverses a list.

List Practice

Here are some example uses (recall that function application does not require parentheses):

```
head [4..6]       -- Returns 4
tail [1..3]       -- Returns [2, 3]
take 3 [1..10]    -- Returns [1, 2, 3]
length "abc"      -- Returns 3. Strings are lists of characters.
length [1..10]    -- Returns 10
product [1..5]    -- Returns 120
reverse [1..5]    -- Returns [5, 4, 3, 2, 1]
```

Some practice questions:

1. How many characters are there in the string "The big bad wolf"? Have Haskell count them!
2. What is the product of all the numbers from 5 to 10?
3. How can we test if a string is a palindrome? (You can use == to compare two strings).

Working with script files, writing functions

While we can write programs in the interactive window, it is easier to write them in a separate script file, then load that file in. To practice this, let us create a new file and load it in:

- Type Ctrl–D to end your current ghci session.
- Create a directory where you will put your Haskell files, and cd to that directory.
- Open up Sublime Text or your favorite editor to that directory. You can typically do this by typing subl . on the terminal.

- Create a new file there with whatever file name you like and the extension .hs. This is not a required extension, but it is customary for Haskell script files.
- Start ghci in the terminal.
- Type :load "yourfilename.hs" to load the module.
- As you make changes to the module in the future, type :reload to have the latest module reloaded based on the most recent version.

Now let us add something to our script. Put in the following two function definitions in the script file and save it:

```
factorial n = product [1..n]

average ns = sum ns `div` length ns
```

Then reload. You should now be able to do average [1..6]. What do you get when you do that? Is that correct?

Notice that these functions do not need a return statement. They are simply an expression, and the result of the function is the result of evaluating the expression.

The definition of the average function above gives us an opportunity to discuss some syntax issues. First note the `div` operator in the middle. There is a function called div, which does integer division of its arguments, like so:

```
div 5 2       -- integer divide 5 by 2
```

Any function of two arguments can instead be written as an operator, inbetween its arguments, by using backticks around the name, like so:

```
5 `div` 2     -- integer divide 5 by 2
```

This is often more natural to read.

The other important feature is that function application does not need any parentheses around the values, and it is also the strongest-binding operation. So in the formula:

```
sum ns `div` length ns
```

The sum of the ns will happen first, then the length of the ns will be computed, and then those two will be used with the `div`. So with parentheses it would have looked like this:

```
(sum ns) `div` (length ns)
```

We also see the syntax for writing functions in a script: You simply write the function followed by the parameters. Then an equal sign, followed by the definition (function body). We can use a where clause to identify parts of the above expression. Do this now, to change the average function to the following, then save and reload:

```
average ns = total `div` count
    where total = sum ns
          count = length ns
```

Function-writing practice

Write functions that accomplish the following:

1. prefix takes a list and returns the first three elements of the list.
2. isPalindrome takes a string and returns whether the string is a palindrome.
3. addInterest takes two arguments: A "principle" amount, and an interest "rate". It returns the new amount if we added the appropriate interest.
4. hasEnough takes a number and a list and returns whether the list has at least that many elements.
5. isDoubled takes a list and returns whether the list is the result of appending a list to itself (in other words, if the first half of the lst is exactly equal to the second half).
6. suffix takes a list and returns the last three elements of the list.