CSCI 239—Discrete Structures of Computer Science

Lab 3—Haskell Lists

This lab will explore lists in Haskell, one of the language's most powerful features.

Objectives:

- to learn to form Haskell lists using square bracket notation and list comprehensions
- to write Haskell functions that build and process lists
- to begin to use built-in Haskell list processing functions

Preliminaries

The sample code for this lab is in the literate Haskell file in the folder for today's lab. Copy the folder `/usr/people/classes/CS239/labs/Lab03_HaskellLists` and its contents into your *CS239* directory. Open the file *CSCI239Lab03.HaskellLists.lhs* in that directory in *gedit* or another text editor to view the sample code. Put your Haskell code for this lab into that file; put answers to exercises in a separate text or Word file.

Part 1: Creating lists in Haskell

Lists are an integral part of the Haskell language. You can specify short simple lists just by putting the elements of the list in square brackets, separated by commas. For example, [1, 7, 8, 2, 5, 7] means a list of six integers, where one is the first element of the list, and seven is both the second and last element of the list.

Enter this list specification at the *ghci* prompt. Haskell will respond with the same list specification. We can see the type of this list with the command "`:type it`". ("it" is the name of the last value returned by the ghci system.) Enter that command, and Haskell will respond with "`it :: [Integer]`". Haskell expresses list types by enclosing the type of the elements of the list in square brackets, so [Integer] means a list of Integer values.

Now enter the following list specifications, and use the "`:type it`" command to see the type of the specified list. Indicate either the type you get or the error message if Haskell didn't accept the list specification. If the result is different than you expect, try to explain why.

1.a [-5, 3, 20, 0, 15, 9, 2]
1.b [3.5, 7, 0, 2, -8.9]
1.c [True, False, True, True]
1.d [True, 2, 7, False]
1.e ['t', 'h', 'i', 's']
1.f ["this", "is", "it"]
1.g ['I', 'am', 'ready']
1.h ['I', "am", "ready"]
1.i ["I", "am", "ready"]
1.j [""]
1.k []
1.l [[1, 2, 3], [4, 5, 6], [7.5, 8.2, 9]]

A few things to note from these exercises: all the elements in a Haskell list must have a common type. *Bool* and *Num* values are distinct, so you can't mix them in a list. Integer literals are polymorphic, so Haskell allows them in a list of floating point values. The character type in Haskell is called *Char*, and character literals are expressed in single quotes; the String type is actually just a synonym for [Char], and lists of characters are typically expressed as String literals enclosed in double quotes. Lists may be empty, and lists may be elements of lists.

Specifying all the elements of a list can be difficult and error-prone if the list is long. Haskell (like Python) provides several tools for building lists. We'll start with range specifications. To specify a list of ordinal values across a range, just put the starting and ending values into the square brackets with two periods between: try [1 .. 10] and ['a' .. 'z']. You can also specify the elements in a range that are a fixed distance apart by listing the first two element and then the range: [0, 2 .. 10] is all the even numbers from zero to ten; [0.0, 0.05 .. 1.0] is all the floating-point numbers from zero to one by 0.05 (but note the floating-point round-off error); [10, 9 .. 1] is the integers from one to ten in descending order.

For the following exercises, use the range construction to specify the indicated lists. Show both your Haskell specification and the ghci output.

- 2.a The integers from negative ten to ten
- 2.b The odd integers from one to nineteen in descending order
- 2.c The letters from 'a' to 'z'
- 2.d The letters from 'A' to 'Z'
- 2.e The characters from 'A' to 'z'; why does the result contain non-letters?
- 2.f The characters from 'a' to 'Z'; why do you get this result?
- 2.g The floating-point numbers from zero to four by thirds; hint: use "`(1/3)`" in the specification; do you get round-off error?

Haskell includes a list concatenation operator (++) that allows you to combine two lists into a single list with the elements of both lists in the same order starting with the first list. For example, [1, 3, 5] ++ [2, 4, 6] returns the list [1, 3, 5, 2, 4, 6]. Since strings are just character lists in Haskell, this operator works for strings too. Use the concatenation operator and range specifications to specify the lists below, again including the Haskell specification and the ghci output.

- 3.a The integers from one to five followed by five down to one
- 3.b The letters of the Latin alphabet, both uppercase and lowercase
- 3.c The letters of the alphabet together with the decimal digits

Note that we've been doing a lot of pattern matching here, and pattern matching is all over in Haskell. (We'll get back to patterns of function parameters in Part 2 of this lab.)

The most powerful list construction tool in Haskell (apart from writing functions) is the list comprehension. A list comprehension looks like a mathematical set notation and constructs a new list from one or more existing lists. Let's start with a simple example: "`[ 2 * n | n <- [0 .. 10] ]`". This list specification can be read as "the list of two times *n* where *n* is taken from the list of values from zero to ten." If you try it in *ghci*, you'll see that it gives the list of even integers from zero to twenty, or exactly the list of values that are two times the values in the list from zero to ten. Here are a few simple exercises; use list comprehensions to complete them:

- 4.a The squares of the numbers from one to twenty (use the '^' operator)
- 4.b The square roots of the numbers from one to ten (use the *sqrt* function)
- 4.c 2^n for numbers from zero to forty
- 4.d The multiples of π/4 for multipliers zero to eight. (*pi* is π in Haskell.)

With list comprehensions, we can also add logical qualifiers that limit the values we take from the source list. For example, suppose we want all the numbers from one to thirty that are not divisible by three. [n | n <- [1 .. 30], n `mod` 3 /= 0] gives the desired result. There can even be more than one qualifier, so [n | n <- [1 .. 50], n `mod` 2 /= 0, n `mod` 3 /= 0] gives the numbers from one to fifty that are neither divisible by two nor by three. (This is a good start on a list of prime numbers.) Specify the following lists and show the results:

- 5.a The floating-point numbers from zero to four by thirds, using (n / 3) in the first part of the comprehension; do you now get less, more or the same round-off error?
- 5.b The numbers from one to twenty that are divisible by two, three or both
- 5.c The letters of the lowercase alphabet that are not vowels

Note that the technique you (should have) used in 5.a was to specify a range of integers and then compute a fractional range by dividing or multiplying the integers by a floating-point constant. This technique is commonly used in for-loops to generate a sequence of evenly-spaced floating-point values:

```
for (int i = 0; i <= 10; i++) {
double x = i / 10.0;
...
```

causes much more accurate results for x than does:

```
for (double x = 0.0; x <= 1.0; x += 0.1) { ...
```

And back to Haskell: There is another data construction in Haskell (besides the list) called the tuple. It consists of two or more data values (not necessarily of the same type) enclosed in parentheses and separated by commas. For example, (3, 4), (5, "five"), and (1.0, 2.8, 4.7) are all tuples with types (Integer, Integer), (Integer, [Char]), and (Double, Double, Double), respectively. We won't look carefully at tuples right now, but we'll use them to construct some potentially useful lists of tuples.

Let's consider the example of constructing the cross product of two lists, that is, a list of tuples containing all pairs of elements from the first list with elements of the second list. We can do this with a list comprehension that specifies tuples for the list elements and takes the elements from the two lists individually. The specification is straightforward: [(n, m) | n <- [0 .. 4], m <- [0 .. 4]]. The only thing new is having two variables in the list specification, and two lists from which they take their values. We get 25 pairs, since there are five elements in each source list. Specify the following lists and show the results:

- 6.a The list of 3-tuples (n, m, nm) where n and m are both taken from the list of numbers from zero to four
- 6.b The list of pairs where the first element of each pair is from the list of numbers from zero to ten, and the second number is the square root of the first
- 6.c The list of pairs where the first element of each pair is from the list of numbers from one to three by 0.1, and the second number is the log of the first

Part 2: Haskell functions that process and produce lists

While the Haskell tools we used above are very powerful, (and we've only scratched the surface,) for some tasks with lists, we need to turn to functions. We'll start with functions that take a list parameter and return a value computed from the list. Here is an example, a function that computes the length of a list:

```
>    count :: Num a => [t] -> a
>    count []      = 0
>    count (x:xs) = 1 + (count xs)
```

A few notes on this function: We used the name "count" because the Haskell Prelude has a built-in "length" function that works in exactly the same way. The type of the function (determined with the `:type` command) is that it takes a list of any type and returns a value of some number type (which may depend on the overall length of the list). The function definition uses pattern matching (notice a pattern here?) for its base case and recursive case; the use of the patterns [] and x:xs here is very common in Haskell functions that deal with lists: every list is either empty ([]), or it consists of a first element (x) followed by zero or more additional elements (xs).

7. Write a Haskell function that computes the sum of a list, using the count function above as a model; check the type with the `:type` command after you have your function working. Why is the type different than count's?
8. Write a simple Haskell function that computes the mean of a list by dividing its sum by its count. What happens when you call the function on an empty list? If you get an error, how do you fix it?

Now let's count the number of times a particular element occurs in a list:

```
>    elemCount :: (Eq a1, Num a) => a1 -> [a1] -> a
>    elemCount elem []     = 0
>    elemCount elem (x:xs)
>      | elem == x         = 1 + (elemCount elem xs)
>      | otherwise         = elemCount elem xs
```

This is a function of two parameters; the first is the element to be counted, and the second is the list to be processed. Notice that the recursive case has two subcases, depending on whether x is the element we're trying to count. You might be tempted to get a little more pattern matching bang for the buck by writing the following:

```
>    elemCount2 elem []        = 0
>    elemCount2 elem (elem:xs) = 1 + (elemCount2 elem xs)
>    elemCount2 elem (x:xs)    = elemCount2 elem xs
```

This code won't compile, however. You can't use pattern matching between parameters to avoid testing whether the two parameters are equal.

9. Write a function vowelCount1 that uses multiple calls to elemCount to count all the vowels (lowercase only) in a string.
10. Write a function vowelCount2 that uses multiple cases to count all the lowercase vowels in a string, with patterns like `'a':xs`.

Test both functions, making sure that both work correctly. Do they have the same types? Which is more efficient? Why?

Now let's write a few functions that return lists. We'll start with a function that mimics the range list constructor:

```
>    range :: (Num a, Ord a) => a -> a -> [a]
>    range n m
>      | n > m     = []
>      | otherwise = n : (range (n + 1) m)
```

Test this function on a few values of numbers. Then try 'a' and 'z'; this doesn't work because addition isn't defined for characters. We can expand the type of this function by replacing “`n + 1`” with “`succ n`”, meaning the successor of or next value after *n*. Now the type becomes “`range :: (Enum a, Ord a) => a -> a -> [a]`”; *Enum* is a more general class than *Num*, so it now works on more type, including characters. (No, you won't be tested on anything in this paragraph.)

11. Write a function that generates the list of squares of numbers from *n* down to one. If *n* is less than one, it should return the empty list; otherwise, it returns *n* squared followed by the list of squares from *n*-1 down to one.
12. Write a function that takes a list of numbers and returns a list of numbers that are the negative of the numbers in the original list. If the original list is empty, it returns the empty list; otherwise, it returns the negative of the first element in the list followed by the list of negatives of the remaining elements in the list.