Dynamic Technology Holdings

meet the λambdas

Workshop II

Coming to Love Lists

Notes & Study Guide

Andrea Falconi
andrea.falconi@gmail.com

June, 2009

Workshop II
Coming to Love Lists

"The conception of list processing as an abstraction created a new world in which designation and dynamic symbolic structure were the defining characteristics. The embedding of the early list processing systems in languages (the IPLs, LISP) is often decried as having been a barrier to the diffusion of list processing techniques throughout programming practice; but it was the vehicle that held the abstraction together."

—Allen Newell and Herbert Simon

Goals.

- ☐ To understand the recursive and polymorphic nature of the list type.
- ☐ To learn how to define functions over lists.

Topics. Polymorphic functions. Cons function and list type. List comprehensions. Pattern matching. Primitive and general recursion over lists. Streams.

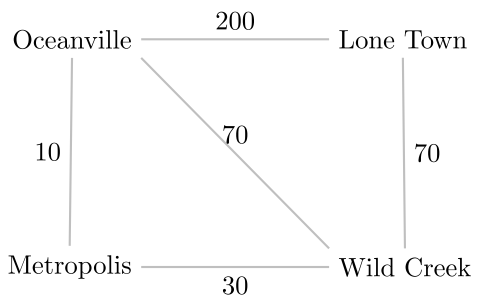
Description. Lists are an essential weapon in the programmer's arsenal and functional programming excels at list processing by providing elegant, concise, yet extremely effective means to compute with lists. The list type is defined in terms of itself (i.e. recursively) and the definition is generic in the sense that it doesn't depend on the type of the contained elements — i.e. it applies as well to integers as to bananas. (Informally, this is what you call polymorphism.) We will construct several lists and let the recursive and polymorphic nature of the list type emerge naturally from the concrete examples. Armed with this insight, we will see how easy it becomes to define functions recursively over lists and, at the same time, keep the definition generic. These functions we will work on are going to be part of a small program that we should be able to run by the end of the session. We will also leave a few minutes to have a further glimpse into the awesome power of lists by hinting at their use for computing with infinite data sequences (aka "streams").

Study Guide.

(1) As usual we will post notes and code on the wiki. Review the material as soon as you can.
(2) Start drilling into list processing by reading §5, §6, and §7 from the *Craft*. It is also a good idea to cross-check your understanding by reading §2 from *RwH*, which covers more or less the same ground.
(3) Practice, practice, practice! Try and do as many exercises as you can from §5, §6, and §7 in the *Craft*. Again, use pencil & paper first and then verify in `ghci`. It is highly recommended that you do the exercises as you read along so that you have a chance to crystallize each concept in your mind before moving on to the next one.
(4) After completing all the reading above, you could optionally read §14 from the *School* to find out more about streams.

Though we can't exactly call ourselves (yet!) seasoned functional programmers, surely we won't say no to something a bit more exciting than the simple program we developed in the past workshop. Yes? Cool, so let's go go!

Today's Source of Headaches. What we're going to develop this time is a program to compute and display the distance chart among a set of cities. We assume there's at least one route between any two cities and we know the distance between any two neighbouring cities, as shown in the following example:

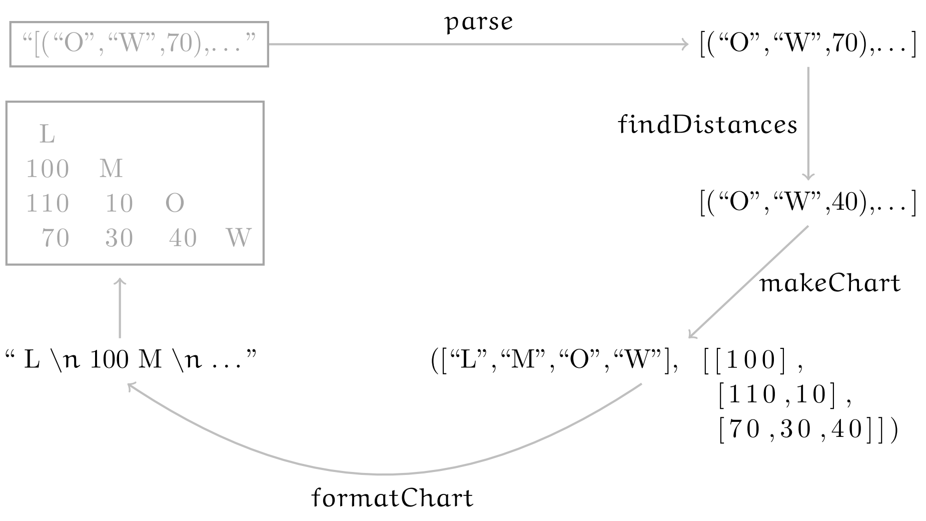


Each straight line represents a road between the cities at the ends of the line, which is also labelled with the length of the road — i.e. the distance between the two cities. Distances are given in kilometers and are positive integers. There may be more than one way from one city to another. For example, if we are in Oceanville and want to go to Wild Creek, we could opt for a 70`km` scenic drive through the countryside along the route that directly connects the two cities; but if we are in a hurry to get there, we'd probably jump on the highway through Metropolis, which is a much shorter drive: $10 + 30 = 40$`km`. Our chart should only give the shortest distance between any two cities and be formatted like the one below which reports the shortest distances among our example's cities.

```
Lone Town
  100  Metropolis
  110    10  Oceanville
   70    30    40  Wild Creek
```

So we want a triangular chart where the city names are given in alphabetical order on the right side (starting from the top) and the distance between two cities is found by the usual mechanism of intersecting rows and columns, so that for example the shortest distance between Lone Town and Oceanville is 110`km`. Distances in each column are right-aligned and we should add enough padding to avoid visual clutter. The city names sit at the end of each row and the first row only contains the name of the first city in alphabetical order. The whole chart should be produced in a string that can be printed on the program's `stdout`. To simplify matters, we're only going to support monospaced fonts and so string formatting becomes easier as we can assume that all characters will have the same width when displayed on screen.

Lunar-Orbit Design View. The problem is not as straightforward as it seems on first read, so to keep our sanity we're going to decompose it into smaller ones that we can more comfortably solve. How to do that? Function pipelines, you say — wow, the brain-washing of the previous workshop must have worked! So here's our high-level decomposition, using the data from the previous example (city names trimmed down to first letter to save space):



The program is going to read a list of triples from its standard input; each triple describes two neighbouring cities and their road distance in kilometers. The list is given just like you would write it in Haskell code, so that the parse function may have an easy time converting the input text into an actual in-memory list of type [(String, String, Int)]. The findDistances function then computes the shortest distance between any two cities (not just neighbouring cities) and produces a list of the same type. Now makeChart can use this information to output the chart data in the form of a pair ([String], [[Int]]), where the first element is the list of all cities in alphabetical order and the second a list of, well, lists of distances — the first list contains the distances for the first row of the chart, the second one corresponds to the second row, and so on. Finally, formatChart assembles all this into a String that we can slap on the program's standard output. Welcome to list processing!

In all likelihood, we'll have to split these functions into smaller ones (yup, more pipeline fun) and so we decide to create a different module for each main function and its subordinates: the Analyzer module will provide the findDistances function, the Chart module makeChart, the Formatter formatChart, and the Main module will pull all the others together as well as providing the code to read the input from `stdin` and write the output to `stdout`. The idea is to confine each design decision in its own module: distance computation (Analyzer), chart creation (Chart), presentation (Formatter), input/output (Main). In other words, we seek an *information hiding* design as formulated by Parnas.[1]

Splitting Nicely. Haskell, like any other language, can assist with the clerical details of modularization such as namespaces, imports/exports, etc. Functions can be grouped into modules, each providing a namespace and a separate compilation unit; you can decide which functions a module may export (i.e. make available to other modules) and which available functions from other modules to use (i.e. import) in a specific module. Each program should contain a module defining the `main` function, which is where evaluation will kick off when the program is run. Here's a self-explanatory cheat sheet:

```
module MyModule where
f1, f2 :: ...
-- all functions (f1 and f2) will be exported

module YourModule (g1, g2) where
g1, g2 :: ...
h1, h2 :: ...
-- only g1 and g2 will be exported

module Main where
import MyModule (f1)      -- f2 will not be visible
import YourModule         -- all exports (g1 and g2)
main = ...                -- program's entry point
```

Each module (usually) goes in its own file with the file name being the same as the module name and an extension of ".hs". To compile the program with `ghc`, go to the directory in which you put the module defining the `main` function (let's say you called that module `Main`) and run: `ghc --make -Wall -o mypgm Main` to produce the executable `mypgm`. This is not the full story, but it's more than enough to keep us going; should you be interested in the gory details, the standard references are [3, 2].

The Main Dude. Back to our program, here's the definition of our Main module:

```
module Main where
import Analyzer
import Chart
import Formatter

parse xs = read xs :: [(String,String,Int)]

main = interact
    (formatChart ∘ makeChart ∘ findDistances ∘ parse)
```

If you're wondering, `read` is a built-in function to deserialize from strings — you only have to specify the type of the serialized data. The `interact` function is also built-in and we use it to pass the data on `stdin` through our pipeline and then output the result to `stdout`.

The Matrix

No, you won't be asked to choose between any red and blue pills, but will be introduced to some rudimentary maths constructions.[1] First off, whenever we have a set (i.e. any collection of objects), we can "build" another one out of it by using a function to transform its elements. For example, if $\mathbb{N}$ is the set of counting numbers, we can build the set D where each element of $\mathbb{N}$ is doubled and we describe it with the following handy notation: $\{2 \times n \mid n \in \mathbb{N}\}$. This simply says that the new set is the collection of objects given by the formula $2 \times n$ where n is an element of $\mathbb{N}$; in fact, the vertical bar symbol means "where" (or "such that") and the funky $\in$ (epsilon) means "is an element of". It is often useful to cherry-pick elements from the original set, for example say we just want to double the even numbers. We specify this by adding a filtering condition like so: $\{2 \times n \mid n \in \mathbb{N}, n \text{ is even}\}$. Another construction you must have come across at least once in your life is that of a (yes, you guessed it) matrix, that is a structured collection of objects arranged in rows and columns.[2] If you have a matrix of 3 rows by 4 columns, how would you describe its indexes? One way is to enumerate explicitly all of them: $\{(1,1),(1,2),\ldots\}$. A better option is to use the above "set builder" notation: $\{(r,c) \mid r \in \mathbb{N}, c \in \mathbb{N}, 1 \leqslant r \leqslant 3 \text{ and } 1 \leqslant c \leqslant 4\}$. So we can see that in general we may want to draw multiple elements from one or more given sets to build a new one.

Basic Comprehension. The way in which we've built new sets out of existing ones is an instance of what mathematicians call *set comprehension*. Most functional programming languages provide a construction identical in concept to build a new list out of existing ones which is called (surprise, surprise) *list comprehension*. Check out how you could recast the above examples in Haskell (for the sake of pretty printing, we'll be using ∈ in place of the two characters `<-` which you'd use to type the program in a conventional text editor):

```
[ 2*n | n∈[0..] ]                         ⤳ [0,2,4,..]
[ 2*n | n∈[0..], n `mod` 2 == 0 ]         ⤳ [0,4,8,..]
[ (r,c) | r∈[1..3], c∈[1..4] ]            ⤳ [(1,1),(1,2),..]
```

If you've followed along, the above should kinda be self-explanatory, provided we clarify the meaning of [0..], [1..3], and [1..4]. Turns out, Haskell let's you easily create a list of integers by using an enumeration syntax:[3] $[m..n]$ stands for the list of integers from m to n so that $[1..3] = [1,2,3]$. If you don't specify the upper bound, then you get an . . . infinite list! For example, $[0..] = [0,1,2,3,..]$ (i.e. all the counting numbers that we denoted above with $\mathbb{N}$); later in the workshop we'll touch on how to deal with such beasties and why they are extremely useful. Back to our examples, we see that the first list comprehension defines a computation that we could informally describe as:

[1] Turns out, the title of a very popular movie is also the name of a very popular maths device!
[2] For what we need to do there's no need to define exactly what a matrix is; instead we'll run with this rather vague definition.
[3] The same trick works for characters and other data types, refer to the *Craft* to find out more.

```
result = new empty list
foreach n in [0..]
    append (2*n) to result
```

That is, to build the result list, iterate through the input list [0..] from its head by taking one element at a time; each time double that element and append the resulting number to the end of the result list. Iteration is the same in the second list comprehension too, except this time we filter out the odd elements from the input list because we only need the even ones to build the result list:

```
result = new empty list
foreach n in [0..]
    if n `mod` 2 == 0
    then append (2*n) to result
```

The last comprehension is a little fancier in that we iterate over two input lists:

```
result = new empty list
foreach r in [1..3]
    foreach c in [1..4]
        append (r,c) to result
```

At this point the question comes up naturally: is this list comprehension thingie just a *for* loop in disguise? The short answer is no. We will revisit all this later on and explain the answer, but for now you may want to think of it in "imperative" terms as we outlined above; it should probably make it easier to absorb the concept initially and so we can quickly move along with the implementation of our program, which will make heavy use of it. However, be prepared for a twist in the tale!

To wrap up with list comprehensions, we see that what goes on the left hand side of the vertical bar is the rule to compute each element of the new list; this computation involves elements coming from one or more input lists, which you specify on the right hand side of the bar. Elements are drawn one at a time from the input list, starting from the head, and processed as specified by the transformation rule on the left of the bar. If there are two input lists, the head of the first list is drawn and then kept fixed while drawing elements from the second list; the procedure is then repeated for the remaining elements of the first list. Optionally, you may specify a filtering condition so that only matching elements are drawn from the input lists.

Charting Away. Time to get cracking with the implementation of our program. Let's start from the Chart module. We have to implement the function makeChart

[(String, String, Int)] $\longrightarrow$ ([String], [[Int]])

[("O","W",40),...] $\longrightarrow$ (["L","M","O","W"], [[100],
[110,10],
[70,30,40]])

which takes a list of distances between cities and produces the corresponding chart data in the form of a pair ([String], [[Int]]), where the first element is the list of all cities in alphabetical order and the second a list of lists of distances — the first list contains the distances for the first row of the chart, the second one corresponds to the second row, and so on. (Oh, forgot to mention, we'll keep on using city names trimmed down to the first letter to save space and as we're at it, we'll stop using quotes as well — hope you won't mind!)

First thing to figure out is how to extract a list of (sorted) cities from the input. To get a clue, we start examining an example input; for instance, this could be what the findDistances function may produce when fed with the distance data for our fictitious cities:

```
[(O, W, 40), (M, O, 10), (M, W, 30),
 (L, M, 100), (L, W, 70), (L, O, 110)]
```

We realize immediately that we can get a list of cities by just iterating this list (using comprehension) and extract the first element out of each triple; similarly we can extract every second element too into another list and these two lists will contain all the possible cities, although with duplicates. So we can join the lists (using the built-in join operator ++), remove the duplicates (using the nub library function), and then sort the list (sort library function) to obtain the list of all cities in alphabetical order. We've just described a function pipeline, which we call extractCities:

$$\begin{tikzcd}
{[(O,W,40),(M,O,10),(M,W,30),(L,M,100),(L,W,70),(L,O,110)]} \arrow[d, "\mathit{listCities}"] \\
{[O, M, M, L, L, L] \mathbin{++} [W, O, W, M, W, O]} \arrow[d, "\mathit{nub}"] \\
{[O, M, L, W]} \arrow[d, "\mathit{sort}"] \\
{[L, M, O, W]}
\end{tikzcd}$$

```
extractCities = sort ∘ nub ∘ listCities
```

To implement it, we only need to define the listCities function ourselves and then import the remaining functions from the Data.List module into our Chart module.[4]

[4] In Haskell, any module implicitly imports all the definition from the Prelude module, which contains most of the library functions we'll see here. Occasionally, we'll import functions from other modules that ship with GHC, in which case, an explicit import statement is required.

```
import Data.List

extractCities = sort ∘ nub ∘ listCities
    where
    listCities zs = [x | (x,_,_)∈zs] ++ [y | (_,y,_)∈zs]
```

Note the weird syntax in the list comprehensions above: $(x, \_, \_)$ and $(\_, y, \_)$. It's just a way to describe what parts of the element being iterated we're actually interested in; to do this we match the element against a pattern describing the data structure and name the parts we want to refer to in our computation. The pattern $(x, \_, \_)$ says: for the triple at hand, bind the name x to whatever value is the first component of the triple; the underscore is just a placeholder for the other components we're not interested in. You can use the same trick for any tuple and we'll soon see that a similar one works for lists too. Indeed, this is an instance of a nifty feature offered by most functional programming languages which goes under the name of *pattern matching*. We'll see many instances of it as we go along.[5]

Let's move on with the implementation of our top-level function, $makeChart$. We still need to produce the distance rows; let's have a look at our example input list again:

```
    [(O, W, 40), (M, O, 10), (M, W, 30),
     (L, M, 100), (L, W, 70), (L, O, 110)]
```

After a little thought we note that distances can be seen as elements of a matrix indexed by city names, taken in alphabetical order — look below.

	L	**M**	**O**	**W**
L	0	100	110	70
M	100	0	10	30
O	110	10	0	40
W	70	30	40	0

How did we figure out which distance goes where in the matrix? Easy, we took each input triple and interpreted, respectively, the first and second components as the row and column indexes. For example, (O,W,40) is telling us to put 40 at position (O,W); because the distance between W and O is obviously still 40, this is also what should go into the matrix cell (W,O).[6] The bottom line is that given any two distinct indexes (x, y) we can look up the corresponding distance in the input list; should we consolidate this train of thought in a function definition? How about

[5] It's probably pointless to formulate the exact rules of pattern matching at this stage; in fact, pattern matching is often best understood by example. However, we'll eventually give a precise definition when we tackle data types in general, at which point we should be able to appreciate the abstract definition fully.

[6] What goes on the main diagonal? Well, each element should obviously be 0 as that is the distance between a city and . . . itself!

```
lookup (x,y) = head [d | (c1,c2,d)∈xs,
                         (x,y) == (c1,c2) ||
                         (x,y) == (c2,c1)]
```

where xs is the input list of triples and $head$ is the built-in library function that returns the head of its input list (e.g. $head\ [1, 2, 3] = 1$) or an error if the list is empty. Because we take (x, y) above from the cities we've previously extracted from xs, it follows that the above comprehension $[d \mid \ldots]$ will always have to result in a list with a least one element[7] and so we can safely use $head$ without risking any, erm . . . runtime "surprises".

Now look at what sits below the main diagonal of the matrix above — numbers in black. Are not those exactly the distances we need for our chart? And they also happen to be arranged in the right order! Well, looks like we've got a promising avenue to explore. We have to collect the numbers on each row (up to the main diagonal) in their own list and then put all those lists together in the chart's distance list: $[[100], [110, 10], [70, 30, 40]]$. Because we already know how to work out distances from matrix indexes (i.e. $lookup$ function), our best bet is to first generate the needed list of indexes and then map each index onto the corresponding distance:

$$\begin{tikzcd} {[L, M, O, W]} \arrow[d, "makeIndexes"] & \\ {[[(M,L)], [(O,L),(O,M)], [(W,L),(W,M),(W,O)]]} \arrow[r, "mapDistance"] & {[[100], [110,10], [70,30,40]]} \end{tikzcd}$$

Cool, let's implement the $mapDistance$ function then. Can we map a single row? Sure we can, simply iterate each matrix index in the row and apply $lookup$; this is an easy job for a list comprehension:

```
mapRow r = [lookup (x,y) | (x,y)∈r]
```

But then we can repeat the trick to map all the rows:

```
mapDistance indexes = [mapRow row | row∈indexes]
```

which can be rewritten by substituting a comprehension for ($mapRow$ row) as defined by the $mapRow$ equation, so we spare a function definition:

```
mapDistance indexes =
            [ [lookup (x,y) | (x,y)∈row] | row∈indexes ]
```

Putting all the pieces together, the definition of $makeChart$ looks like this:

[7] Would you be able to *prove* it? Give it a try.

```
makeChart :: [(String,String,Int)] → ([String],[[Int]])
makeChart xs = (cities,distances)
  where
  cities = extractCities xs
  distances = (mapDistance ∘ makeIndexes) cities

  mapDistance indexes =
                  [ [lookup (x,y) | (x,y)∈row] | row∈indexes ]
  lookup (x,y) = head [d | (c1,c2,d)∈xs,
                                (x,y) == (c1,c2) ||
                                (x,y) == (c2,c1)]
```

A Hard Nut. Finally, the only thing left to complete the implementation of the Chart module is the definition of *makeIndexes*. Earlier on we saw how to use a list comprehension to generate all the indexes of a matrix; looking again at the matrix above, we see that in our case we only need to take those indexes where the row index is greater than the column index — e.g. (M,L), M comes after L in the alphabetical order. This is an easy enough tweak:

```
[ (x,y) | x∈cities , y∈cities , x > y ]
```

which will work because the elements of *cities* are ordered and Haskell has built-in string comparison operating on the alphabetical order. However, although this list contains all the indexes we need and in the correct order, we still have to collect the indexes in lists, one in correspondence of each matrix row. In other words, we should define a function to carry out transformations similar to the one below:

$$\begin{array}{l}[(M,L),\\ \ (O,L),(O,M),\\ \ (W,L),(W,M),(W,O)]\end{array} \xrightarrow{\mathit{makeRows}} \begin{array}{l}[[(M,L)],\\ \ [(O,L),(O,M)],\\ \ [(W,L),(W,M),(W,O)]]\end{array}$$

The first observation we make is that we have to collect elements from the input list in the same order in which they are given. What makes it awkward though is the fact that we have to collect them in different lists and it's a different number every time. However, the example is actually giving us a clue of how to proceed. In fact, we see that to build the first row, we need to take the first element (M,L) out of the input list and create a new list containing just (M,L); this leaves us with a smaller list: [(O,L),(O,M),(W,L),(W,M),(W,O)]. From this smaller list we have to take out the first two elements (O,L) and (O,M) to create a new list for the second row. At this point we're left with [(W,L),(W,M),(W,O)] and we have to take the three elements out to form the list for the third row. As we go along, we have to add these new lists to the result list.

To help us with the task of taking and discarding elements from a list, we turn to two handy library functions: *take* and *drop*. The first one extracts a sublist of a given length from an input list, starting from the head — e.g. $\mathit{take}\ 2\ [1,2,3] = [1,2]$.

The second one returns the sublist obtained from the input list by removing the specified number of elements, again starting from the head — e.g. $\mathsf{drop}\ 2\ [1,2,3] = [3]$. Here's a diagram to summarize our plan of action:

$$\begin{tikzcd}
\begin{array}{l}[\,\boxed{(M,L)}\,,\\ (O,L)\,,(O,M)\,,\\ (W,L)\,,(W,M)\,,(W,O)\,]\end{array} \arrow[r, "\mathsf{take}\ 1"] \arrow[d, "\mathsf{drop}\ 1"'] & [\,\underline{(M,L)}\,] \\
\begin{array}{l}[\,\boxed{(O,L)\,,(O,M)}\,,\\ (W,L)\,,(W,M)\,,(W,O)\,]\end{array} \arrow[r, "\mathsf{take}\ 2"] \arrow[d, "\mathsf{drop}\ 2"'] & [\,(M,L)\,]\ :\ [\,\underline{(O,L)\,,(O,M)}\,] \\
[\,\boxed{(W,L)\,,(W,M)\,,(W,O)}\,] \arrow[r, "\mathsf{take}\ 3"] & \begin{array}{c}[\,(M,L)\,]\ :\ [\,(O,L)\,,(O,M)\,]\ :\\ [\,\underline{(W,L)\,,(W,M)\,,(W,O)}\,]\end{array}
\end{tikzcd}$$

The result list in this case is given by:

$$\mathsf{take}\ 1\ xs\ :\ \mathsf{take}\ 2\ (\mathsf{drop}\ 1\ xs)\ :\ \mathsf{take}\ 3\ (\mathsf{drop}\ 2\ ys)$$

where xs is the initial input list and $ys = \mathsf{drop}\ 1\ xs$. How do we generalize all this to the case of an arbitrary number of cities? That is, looking at the example we have a concrete solution for the case of 4 cities (and hence 3 rows and 6 matrix indexes), but how can we give a generic definition of the $\mathsf{makeRows}$ function?

Help! Sometimes, when facing a tough nut, it may help to first solve similar but simpler problems which will hopefully teach us something about the original problem. If we just had to turn a flat list into a list of singleton lists (i.e. each containing one element from the original list), then things wouldn't be that difficult — e.g. $[1,2,3] \rightsquigarrow [[1],[2],[3]]$. In fact, we can promptly use our template for recursive definitions:

- ☐ If the input list is empty the result is the empty list too: $\varphi\ [\,] = [\,]$.
- ☐ If we take an arbitrary list xs and we assume $\varphi\ xs$ yields a list of singleton lists, then we can define $\varphi\ (x : xs) = [x]\ :\ \varphi\ xs$.

Now it's easy enough to throw the take and drop functions in the mix as $\mathsf{take}\ 1\ (x : xs) = [x]$ and $\mathsf{drop}\ 1\ (x : xs) = xs$; so said $zs = (x : xs)$, with simple substitutions we get to

```
φ  [ ]  =  [ ]
φ  zs  =  take  1  zs  :  φ  (drop  1  zs)
```

Of course, if we were given a list of $2 \times k$ elements and wanted to extract two elements at a time, we could use exactly the same function definition, provided we substitute 2 for 1. This is telling us that we can trivially generalize the definition of φ to accept the number of elements to extract in each list as a parameter n:

```
φ n [] = []
φ n zs = take n zs : φ n (drop n zs)
```

So we see that in general if we're given a flat list of $n \times k$ elements we can turn it into a list of lists by taking the first n elements from the input list and then repeat the procedure on the list obtained from the input list by dropping the first n elements. Eventually we'll end up removing all elements from the input list, at which point the first equation tells us to stop the recursion and return the empty list. Here's an evaluation example:[8]

$$\begin{aligned}
&\phi\ 2\ [1,2,3,4] \\
&\qquad = \mathit{take}\ 2\ [1,2,3,4]\ :\ \phi\ 2\ (\mathit{drop}\ 2\ [1,2,3,4]) \\
&\qquad = [1,2]\ :\ \phi\ 2\ (\mathit{drop}\ 2\ [1,2,3,4]) \\
&\qquad = [1,2]\ :\ \phi\ 2\ [3,4] \\
&\qquad = [1,2]\ :\ \mathit{take}\ 2\ [3,4]\ :\ \phi\ 2\ (\mathit{drop}\ 2\ [3,4]) \\
&\qquad = [1,2]\ :\ [3,4]\ :\ \phi\ 2\ (\mathit{drop}\ 2\ [3,4]) \\
&\qquad = [1,2]\ :\ [3,4]\ :\ \phi\ 2\ [\,] \\
&\qquad = [1,2]\ :\ [3,4]\ :\ [\,] \\
&\qquad = [[1,2],[3,4]]
\end{aligned}$$

Great. So we can turn a flat list into a list of lists by collecting a fixed number of elements at each computation step. If only we could specify a variable number of elements to collect, we'd have arrived at the definition of $\mathit{makeRows}$. How does the number of elements to collect at each step vary in the case of $\mathit{makeRows}$? Looking at the above example again,

$$\mathit{take}\ 1\ xs\ :\ \mathit{take}\ 2\ (\mathit{drop}\ 1\ xs)\ :\ \mathit{take}\ 3\ (\mathit{drop}\ 2\ ys)$$

we see that we collect 1 element at the first step, $2 = 1 + 1$ elements at the second, $3 = 2 + 1$ at the third, and so on. This suggests that at step n we should take n elements and then call $\mathit{makeRow}$ to collect $n + 1$ elements out of the list obtained by dropping n elements from the list at hand. But then, we can easily tweak ϕ to obtain $\mathit{makeRows}$:

```
makeRows n [] = []
makeRows n zs = take n zs : makeRows (n+1) (drop n zs)
```

and in fact, if we evaluate it[9]

[8] To save space we evaluate some of the expressions eagerly.
[9] Again, to save space we evaluate some of the expressions eagerly.

$$
\begin{aligned}
&\mathit{makeRows}\ 1\ [1,2,3,4,5,6]\\
&= \mathit{take}\ 1\ [1,2,3,4,5,6]\ :\ \mathit{makeRows}\ 2\ (\mathit{drop}\ 1\ [1,2,3,4,5,6])\\
&= [1]\ :\ \mathit{makeRows}\ 2\ (\mathit{drop}\ 1\ [1,2,3,4,5,6])\\
&= [1]\ :\ \mathit{makeRows}\ 2\ [2,3,4,5,6]\\
&= [1]\ :\ \mathit{take}\ 2\ [2,3,4,5,6]\ :\ \mathit{makeRows}\ 3\ (\mathit{drop}\ 2\ [2,3,4,5,6])\\
&= [1]\ :\ [2,3]\ :\ \mathit{makeRows}\ 3\ (\mathit{drop}\ 2\ [2,3,4,5,6])\\
&= [1]\ :\ [2,3]\ :\ \mathit{makeRows}\ 3\ [4,5,6]\\
&= [1]\ :\ [2,3]\ :\ \mathit{take}\ 3\ [4,5,6]\ :\ \mathit{makeRows}\ 4\ (\mathit{drop}\ 3\ [4,5,6])\\
&= [1]\ :\ [2,3]\ :\ [4,5,6]\ :\ \mathit{makeRows}\ 4\ (\mathit{drop}\ 3\ [4,5,6])\\
&= [1]\ :\ [2,3]\ :\ [4,5,6]\ :\ \mathit{makeRows}\ 4\ [\,]\\
&= [1]\ :\ [2,3]\ :\ [4,5,6]\ :\ [\,]\\
&= [[1],[2,3],[4,5,6]]
\end{aligned}
$$

Finally we're in a position to give the definition of $\mathit{makeIndexes}$ and so complete the implementation of the Chart module.

```
makeIndexes cities = makeRows 1 flatIndexes
    where
    flatIndexes = [ (x,y) | x∈cities , y∈cities , x > y ]

    makeRows n [] = []
    makeRows n xs = take n xs : makeRows (n+1) (drop n xs)
```

Awesome!

Bending the Rules. You may have come across the term *primitive* recursion over lists. It is indeed the name under which goes the recursion template we gave in the last workshop.[10] In a nutshell, to define a function f over a list using primitive recursion:

- ☐ Give the definition on [ ]. (This is called the *base case.*)
- ☐ Take an arbitrary non-empty list $(x : xs)$ and define f in terms of x, xs, and $f\ xs$.

Although this scheme serves us well for most cases, sometimes we need to bend the rules as we did earlier on. Here's a more general scheme then, which captures the pattern we used to define the $\mathit{makeRows}$ function:

- ☐ Give the definition on [ ]. (The *base case.*)
- ☐ Take an arbitrary list xs and define f in terms of xs and $f\ ys$, where ys is a *smaller* list obtained from xs.

[10] Actually, our initial template is a specialization of the primitive recursion scheme we're giving here, but we're gonna leave the subtleties for another day. . .

A further variation on the theme is to establish the base case on all lists of a given length instead of defining the function on the empty list. For example, say we want to define a function to work out the maximum of a list of integers. We've learnt how to "fold" operators into a list to obtain a single value and so we can use the same procedure in this case with the built-in max operator: 2 ‘max‘ 3 ‘max‘ $0 = 3$. If we used primitive recursion, we'd have to associate an integer to the empty list, but which one? We could try

```
foldMax     []    = 0
foldMax   (x:xs)  = x  `max`  foldMax  xs
```

but this will not work if the list contains negative integers:

$$\mathsf{foldMax}\ [-1] = -1\ \text{‘}\mathsf{max}\text{‘}\ 0 = 0$$

Indeed, we have no sensible value to associate to the empty list. A way out then is not to define the function on the empty list at all, and rather give the definition on lists of length $n \geqslant 1$. We take a generic list of length 1 so that our base case applies to *all* lists of length 1 — not just a particular one, like [3] or [−2].

```
foldMax  [x]      = x
foldMax  (x:xs)  = x  `max`  foldMax  xs
```

However, this leads to a "Catch-22" situation where to define the function we don't consider all possible inputs (e.g. [ ] in the case of foldMax) and so according to our definition of a function ... we're not defining a function! Such a beastie is called a *partial function* because the rule is only defined *for some* values of the input type but *not for all*. Be very careful when using partial functions as the program could easily bomb out; in fact, the Haskell runtime aborts execution if asked to evaluate a partial function on a value for which it was not defined. Consider a function to convert the maximum of a list to a string:[11]

```
showMax  xs  =  show  (foldMax  xs)
```

If you accidentally pass the empty list to showMax then you get a runtime error and the program terminates.[12] Probably when defining the showMax function we forgot that foldMax is not defined on the empty list and so ... oh well, you know the story. (Think `null` pointers.)

[11] The built-in function show converts a value to a string.

[12] Yes, you can "catch" errors, but we'll deal with that in another workshop.

Comprehending Comprehensions

We've been thinking of list comprehensions as a sort of *for* loop and that's helped us click on how to write code that iterates the elements of a list. However, the analogy only stretches so far and there are essential differences between the two that actually make them very diverse species. In fact, a comprehension does not impose a termination condition on the computation as the *for* loop does — i.e. `for i=0 to 10`. This is a consequence of the fact that list comprehension is just *syntactic sugar* for the application of filtering and mapping functions to a list! In turn, this implies that the same rules of lazy evaluation apply. In a nutshell, list comprehensions do *not* add anything new to the functional world we've been describing so far, they are just a convenient way to apply a mapping and (possibly) filtering function to a list. Let's have a look at all this in more detail.

Desugaring. We began explaining list comprehension with a few simple examples:

```
[ 2*n | n∈[0..] ]                        ⇝ [0,2,4,..]
[ 2*n | n∈[0..], n `mod` 2 == 0 ]        ⇝ [0,4,8,..]
[ (r,c) | r∈[1..3], c∈[1..4] ]           ⇝ [(1,1),(1,2),..]
```

Pretend we didn't know about list comprehensions, how could we go about implementing the above computations? The first one seems to be pretty trivial: double each element in the list of positive integers. All we need to do then is come up with a function that given a list of integers doubles each element. Trivially, we can define a function to double an integer: $\mathsf{double}\ n = 2 * n$; so all we need to do is define a function $\mathsf{mapDouble}$ that given a list transforms each element using the double function. If the input list is empty, there's nothing to do and so $\mathsf{mapDouble}\ [\,] = [\,]$. Using our recursion template, we may assume that we have a definition for $\mathsf{mapDouble}\ xs$, xs being an arbitrary list of integers — e.g. we assume that $\mathsf{mapDouble}\ [0, 1, 2] = [\mathsf{double}\ 0, \mathsf{double}\ 1, \mathsf{double}\ 2]$. Can we give a definition for $x : xs$ relying on the fact that $\mathsf{mapDouble}\ xs$ "works"? Trivial, you say.

```
mapDouble    []    = []
mapDouble  (x:xs)  = double x : mapDouble xs
```

Almost without any further thinking we see that:

```
[ 2*n | n∈[0..] ] = mapDouble [0..]
```

But this suggests that we could get the second comprehension pretty similarly, if only we could chop off all the odd numbers from the list before passing it on to $\mathsf{mapDouble}$. In other words, if we could define a function ϕ to filter odd elements out of a given list of integers (e.g. $\phi\ [0, 1, 2] = [0, 2]$), then trivially

```
[ 2*n | n∈[0..], n `mod` 2 == 0 ] = mapDouble (φ [0..])
```

How do we define ϕ though? What we've just done for $\mathsf{mapDouble}$ is to regard the expression $2 * n$ as defining a function, namely the double function. What is n `mod` $2 == 0$? A function that given an integer n returns True if the integer is even, False otherwise. Of course! We define: $\mathsf{isEven}\ n = n$ `mod` $2 == 0$. Now we can go through a list of integers and collect those for which isEven returns True, while we discard the others. At this point it's clear that ϕ is to be a function that uses isEven as a filtering criteria; so we rename ϕ to something more meaningful, such as $\mathsf{filterIsEven}$. Again, to define it, we turn to our template for recursive definitions. If the input list is empty, $\mathsf{filterIsEven}\ [\,] = [\,]$. This way we know that the resulting list will contain no odd numbers. Make the same assumption for a generic list of integers xs — e.g. $\mathsf{filterIsEven}\ [1, 2, 3] = [2]$; how can we give a sound definition for $x : xs$ using this assumption? If you haven't guessed already, look below.

```
filterIsEven       [] = []
filterIsEven  (x:xs)
     | isEven x      = x : filterIsEven xs
     | otherwise     =     filterIsEven xs
```

If we replace the mapping and filtering expressions in the original comprehensions with the corresponding functions, we can wrap up on all the above like so:

```
[ double n | n∈[0..] ] = mapDouble [0..]

[ double n | n∈[0..], isEven n ] =
                          mapDouble (filterIsEven [0..])
```

A moment of reflection and we see that a pattern begins to emerge. The expression on the left of the comprehension vertical bar, can be regarded as a function t to transform each element of the input list. The condition on the far right (if there's any) can be regarded as a function p to test whether or not the element at hand should be passed on to the transformation function and so be part of the final list. We can always write a mapping function mapT that applies t to each element of a list and, by the same token, a function $\mathsf{filterP}$ that filters out of the list those elements that don't pass the test p. Therefore, in general, we always have:

```
[ t x | x∈xs, p x ] = (mapT ∘ filterP) xs
```

In other words, a list comprehension with one input list (aka *generator*) is just syntactic sugar for the application of (possibly) a filtering and then a mapping function to the input list! What happens when we're drawing elements from multiple lists, like in the third comprehension example above? We have a similar situation, except we need to throw the concat library function in the mix. This function flattens a lists of lists — e.g. $\mathsf{concat}[[1, 2], [3]] = [1, 2, 3]$. Here's how you would "desugar" the last comprehension above then.

```
comp = [ (r,c) | r∈[1..3], c∈[1..4] ]

mapPair x     []  = []
mapPair x (y:ys)  = (x,y) : mapPair x ys
```

```
mapMapPair     []    ys = []
mapMapPair  (x:xs)  ys = mapPair x ys : mapMapPair xs ys

comp' = concat (mapMapPair [1..3] [1..4])

comp == comp'         ⇝ True
```

We leave it to you to puzzle this out!

Wrapping Up. TODO

* not a for loop, termination condition separated from loop body

* lazy eval applies; example of lookup function: efficient!

* general rules for comprehension

Your Turn. TODO

* point reader to ChartNoComp module

* pipeline pix for Formatter (solution outline)

* ask reader to try and implement it with list comprehensions — solution in Formatter.hs

* then they should turn all comprehensions in equiv map+filter expr

The *Cons*-ummate List

TODO

Ignorance May Be Bliss. TODO

* define a few polymorphic functions and note how def applies to several types

* define concept

She Has it All. TODO

* look back and show how to build lists we've been dealing with

* point out recursion and polymorphism

* : and [] are syntactic sugar, expain: List a = Nil | Cons a (List a)

The Condensed Ideas

TODO: sum up what is essential to remember

References

[1] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[2] S. Peyton-Jones et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[3] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.