What, no iteration or loops?

- You may have noticed that in our “Getting Started” tour of Haskell there was no mention of *for* or *while* loops.
- That’s right. There are no explicit iteration or looping operators in built in to Haskell.
- But surely iteration and loops are fundamental operations in programming, can we do without them?
- Yes, sort of, we have other constructs that can take their place: recursion can easily be coded to implement a “**while**” loop (similarly for a *for* loop):

```haskell
whyle x res | b = let res = e in whyle (x+1) res
             | otherwise = res
```

Where *b* is the guard and *e* is the body of the loop

But why would you want to do that?
Special Iteration

• One kind of iteration that we do an awful lot of in Haskell is to walk across a list.
• This can be done using a recursive function easily enough.
• But Haskell provides a more convenient form for doing this very similar to Java enhanced loops:

  for (Object element : anIterableObject)

• Haskell takes Lists (obvs) as the iterable structures and the construct is called **List Comprehension**
• List comprehensions are syntactic sugar in Haskell
• We write a List Comprehension in a style that should be familiar to you from Foundations of CS
Set Comprehension

• In mathematics we use the set comprehension notation to construct new sets from existing sets.

\[ \{ x^2 \mid x \in \{1, \ldots, 5\} \} \]

• This is the set \( \{ 1, 4, 9, 16, 25 \} \)

• Effectively, we ‘iterate’ across the set \( \{1,2, 3, 4, 5\} \) and apply the operation given to the left of the “such that” symbol (i.e. \( \mid \)).

• This can be done for any set, including infinite sets,

\[ \{ x \mid x \mod 2 = 0 \text{ and } x \in \mathbb{N} \} \]

• Defines the set of all even natural numbers
List Comprehension

- Haskell likes to stay as close to the mathematics as it can so the syntax for List comprehension is very similar.

\[ [ x^2 | x <- [1..5] ] \]

- Produces the list of values \([1,4,9,16,25]\)!

\[ [ x | x <- [0..] , x \mod 2 == 0 ] \]

- Produces the infinite list of even naturals

- Here we have used the enumeration notation \([n..m]\) as the generator.

- In general, any expression that returns a list can be used as a generator.
Multiple Generators

• List comprehensions may contain multiple generators.
• For example,

\[
> \left[ (x, y) \mid x \leftarrow [1, 2, 3] , y \leftarrow [4, 5] \right] \\
\left[ (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5) \right]
\]

• The order the in which the generators appear is significant.
• For example,

\[
> \left[ (x, y) \mid y \leftarrow [4, 5] , x \leftarrow [1, 2, 3] \right] \\
\left[ (1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5) \right]
\]

• Think *nested loops*, with later generators as more deeply nested loops whose variables change value more frequently.
Multiple Generator Example

> \[(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]\]

\[\{(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)\}\]

\(x \leftarrow [1, 2, 3]\) is the last generator, so the value of the x component of each pair changes most frequently.
Dependent Generators

- Later generators can depend on the variables that are declared in earlier generators

\[
> \left\{ (x,y) \mid x \leftarrow [1..3] , y \leftarrow [x..3] \right\} \\
\{ (1,1), (1,2), (1,3), (2,2), (2,3), (3,3) \}
\]

- Obviously the order of the generators is very important here

\[
> \left\{ (x,y) \mid y \leftarrow [x..3] , x \leftarrow [1..3] \right\} \\
\text{error: Variable not in scope: x}
\]

- Nice example: flatten a list of lists to a single list

\[
\text{flatten} :: [[a]] \rightarrow [a] \\
\text{flatten xss} = [ x \mid xs \leftarrow xss , x \leftarrow xs ]
\]
Guards in List Comprehensions

- List comprehensions may use guards to filter out the values produced by the generators.

\[
[x \mid x \leftarrow [1..10], \text{even } x]
\]

\[
[2, 4, 6, 8, 10]
\]

- Again, variables used in guards must be declared before their use.

\[
[x \mid \text{even } x, x \leftarrow [1..10]]
\]

error: Variable not in scope: x
Guards example

```haskell
factors :: Int → [Int]
factors n = [ x | x <- [1..n], n `mod` x == 0 ]

> factors 15
[1,3,5,15]

prime :: Int → Bool
prime n = factors n == [1,n]

> prime 15
False

primesUpto :: Int → [Int]
primesUpto n = [ x | x <- [0..n], prime x ]

> primesUpto 30
[2,3,5,7,11,13,17,19,23,29]
```
Zip-a-Dee-Doo-Dah

• Occasionally it is useful to iterate across two structures simultaneously.

• In Haskell the typically way of doing this is to \texttt{zip} the structures together.

\[
\text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]
\]

\[
\text{zip \ [\] \ _ = \ []}
\]

\[
\text{zip \ _ \ [] = \ []}
\]

\[
\text{zip \ (x:xs) \ (y:ys) = (x,y) : zip \ xs \ ys}
\]

• So we take two lists and form a single list of pairs.

• We can then use a list comprehension to iterate over this list of pairs.
Using Zips

• Let’s build an example where we combine zips and List comprehensions.

• First, we’ll use zip to return a list of pairs of adjacent elements in a list

  \[
  \text{pairs} :: [a] \rightarrow [(a,a)] \\
  \text{pairs xs} = \text{zip xs (tail xs)}
  \]

• This is useful for using a list comprehension to define a predicate to check whether a list is sorted.

  \[
  \text{sorted} :: \text{Ord a} \Rightarrow [a] \rightarrow \text{Bool} \\
  \text{sorted xs} = \text{and} [ x \leq y \mid (x,y) \leftarrow \text{pairs xs} ]
  \]
Another Zippy example

• Here is a function that returns the list of all index positions of a given value in a list:

```haskell
positions :: Eq a => a → [a] → [Int]
positions x xs = [ index | (x',index) <- zip xs [0..], x==x' ]
```

- It uses `zip`, list comprehension and guards to achieve this.
- Note that in the previous two examples we used an interesting form of List Comprehension.

```haskell
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```
List Comprehensions - General Form

• The general form for List Comprehensions is actually as follows:

\[
\left[ \text{exp} \mid \text{qual}_1, \text{qual}_2, \ldots, \text{qual}_n \right]
\]

• Where \text{exp} is any expressions and

• \text{qual} is one of

  • \text{pattern} <- \text{exp} (to iterate over the list \text{exp} and pattern match bind the result to \text{pattern})
  
  • \text{exp} where a guard is needed
  
  • \text{let} declaration - for locally declared values

• For example

\[
\begin{align*}
\text{heads} &:: [[a]] \rightarrow [a] \\
\text{heads} \ xss &\ = \left[ x \mid (x : _) \ <- \ xss \right]
\end{align*}
\]

\[
> \text{heads} \ [ [1,2],[3,4,5],[],[8]] = [1,3,8]
\]
String Comprehensions

• Recall that a string is just a List of characters.
• Hence we can use list comprehensions to iterate across strings in the obvious way.
• For example

```haskell
count :: Char -> String -> Int
count x xs = length [ x' | x' <- xs , x==x' ]
```

> count ‘s’ "Haskell is rubbish"
3
A note on Enumerations

• We call the notation \([n..m]\) etc an enumeration
• The notation is sugar for calls to functions from the Enum class.
• As such, the notation supports expressions of any type that are instances of Enum
• This includes all Num types, Bool, Enumerated types, and Char.
• So we can write \(\text{[a’..e’]}\) for the string “abcde”
• There is also a special form \([n,n’..m]\) called “from, then, to”
• This calculates the “difference” from \(n\) to \(n’\) and increments in those steps until \(m\) is reached

\[
> \text{[0,12..60]}\\
\text{[0,12,24,36,48,60]}
\]

\[
> \text{[a’,’c’..’z’]}\\
\text{“acegikmoqsuwy”}
\]
## Quick Quiz - Comprehension

Rewrite the following in simpler form

<table>
<thead>
<tr>
<th>Expression</th>
<th>Simpler</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 2\cdot x + 1 \mid x \leftarrow [0..] ]</td>
<td>???</td>
</tr>
<tr>
<td>[ \chr x \mid x \leftarrow [97..122] ]</td>
<td>???</td>
</tr>
<tr>
<td>sum [ 1 \mid x \leftarrow xs ]</td>
<td>???</td>
</tr>
<tr>
<td>and [ x==y \mid (x,y) \leftarrow \text{zip} ; xs ; (\text{reverse} ; xs) ]</td>
<td>???</td>
</tr>
<tr>
<td>concat [ [ (x,y) \mid y \leftarrow [3,4] ] \mid x \leftarrow [1,2] ]</td>
<td>???</td>
</tr>
<tr>
<td>[ x \mid \text{let} ; ps = (\text{primesUpto} ; n) ; ++ ; [n+1], ; (p,q) \leftarrow \text{zip} ; ps ; (\text{tail} ; ps), ; x \leftarrow [p+1,q-1] ]</td>
<td>???</td>
</tr>
</tbody>
</table>
YOUR QUESTIONS

Next Lecture :
Recursion