List Comprehensions

Haskell has a notation called list comprehension (adapted from mathematics where it is used to construct sets) that is very convenient to describe certain kinds of lists. Syntax:

\[
[ \text{expr} \mid \text{qualifier}, \text{qualifier}, \ldots ]
\]

In English, this reads:

“Generate a list where the elements are of the form \text{expr}, such that the elements fulfill the conditions in the \text{qualifier}s.”

The expression can be any valid Haskell expression.

The qualifier\text{s} can have three different forms: Generators, Filters, and Local Definitions.
Generator Qualifiers

- Generate a number of elements that can be used in the expression part of the list comprehension. Syntax:
  
  \[ \text{pattern} \leftarrow \text{list_expr} \]

- The pattern is often a simple variable. The list_expr is often an arithmetic sequence.

\[
[n \mid n \leftarrow [1..5]] \Rightarrow [1, 2, 3, 4, 5]
\]

\[
[n \ast n \mid n \leftarrow [1..5]] \Rightarrow [1, 4, 9, 16, 25]
\]

\[
[(n, n \ast n) \mid n \leftarrow [1..3]] \Rightarrow [(1, 1), (2, 4), (3, 9)]
\]
A filter is a boolean expression that removes elements that would otherwise have been included in the list comprehension. We often use a generator to produce a sequence of elements, and a filter to remove elements which are not needed.

\[ [n \times n \mid n \leftarrow [1 \ldots 9], \text{even } n] \Rightarrow [4, 16, 36, 64] \]

\[ [(n, n \times n) \mid n \leftarrow [1 \ldots 3], n < n \times n] \Rightarrow [(2, 4), (3, 9)] \]
We can define a **local variable** within the list comprehension. Example:

\[
[n \times n \mid n = 2] \Rightarrow [4]
\]
Qualifiers

Earlier generators (those to the left) vary more slowly than later ones. Compare nested for-loops in procedural languages, where earlier (outer) loop indexes vary more slowly than later (inner) ones.

Pascal:

```
for i := 1 to 9 do
    for j := 1 to 3 do
        print (i, j)
```

Haskell:

```
[(i,j) | i<-[1..9], j<-[1..3]] ⇒
[ (1,1), (1,2), (1,3),
  (2,1), (2,2), (2,3),
  ...
  (9,1), (9,2), (9,3) ]
```
Qualifiers... 

Qualifiers to the right may use values generated by qualifiers to the left. Compare Pascal where inner loops may use index values generated by outer loops.

**Pascal:**

```pascal
for i := 1 to 3 do
    for j := i to 4 do
        print (i, j)
```

**Haskell:**

```haskell
[(i,j) | i<-[1..3], j<-[i..4]] ⇒
[ (1,1), (1,2), (1,3), (1,4),
  (2,2), (2,3), (2,4),
  (3,3), (3,4) ]

[n*n | n<-[1..10], even n] ⇒ [4,16,36,64,100]
```
Define a function `doublePos xs` that doubles the positive elements in a list of integers.

**In English:**

“Generate a list of elements of the form \(2 \times x\), where the \(x\):s are the positive elements from the list \(xs\).

**In Haskell:**

```haskell
doublePos :: [Int] -> [Int]
doublePos xs = [2*x | x<-xs, x>0]
```

> `doublePos [-1,-2,1,2,3]`
> 
> `[2,4,6]`

Note that \(xs\) is a list-valued expression.
Define a function \texttt{spaces \ n} which returns a string of \texttt{n} spaces.

\textbf{Example:}

\begin{verbatim}
> spaces 10
" "
\end{verbatim}

\textbf{Haskell:}

\begin{verbatim}
spaces :: Int -> String
spaces n = [' ' | i <- [1..n]]
\end{verbatim}

\begin{itemize}
  \item Note that the expression part of the comprehension is of type \texttt{Char}.
  \item Note that the generated values of \texttt{i} are never used.
\end{itemize}
Example

Define a function \texttt{factors \ n} which returns a list of the integers that divide \(n\). Omit the trivial factors 1 and \(n\).

**Examples:**

\[
\begin{align*}
factors\ 5 \Rightarrow [ ] \\
factors\ 100 \Rightarrow [2, 4, 5, 10, 20, 25, 50] \\
\end{align*}
\]

\textbf{In Haskell:}

\[
\begin{align*}
factors \ :: \ \text{Int} \rightarrow [\text{Int}] \\
factors\ n = [i \mid i<-[2..n-1], n \mod i == 0]
\end{align*}
\]
Example

Pythagorean Triads:

• Generate a list of triples \((x, y, z)\) such that \(x^2 + y^2 = z^2\) and \(x, y, z \leq n\).

\[
\text{triads } n = [(x, y, z) | \\
x<-[1..n], y<-[1..n], z<-[1..n], \\
x^2 + y^2 == z^2]
\]

\[
\text{triads } 5 \Rightarrow [(3, 4, 5), (4, 3, 5)]
\]
We can easily avoid generating duplicates:

\[
\text{triads' } n = [(x, y, z) | \\
x \leftarrow [1..n], \; y \leftarrow [x..n], \; z \leftarrow [y..n], \\
x^2 + y^2 == z^2] 
\]

\[
\text{triads' } 11 \Rightarrow [(3,4,5), \; (6,8,10)] 
\]
Example – Making Change

Write a function `change` that computes the optimal (smallest) set of coins to make up a certain amount.

Defining available (UK) coins:

```haskell
type Coin = Int
coins :: [Coin]
coins = reverse (sort [1,2,5,10,20,50,100])
```

Example:

```
> change 23
[20,2,1]
> coins
[100,50,20,10,5,2,1]
> all_change 4
[[2,2],[2,1,1],[1,2,1],[1,1,2],[1,1,1,1]]
```
Example – Making Change...

- **all_change** returns all the possible ways of combining coins to make a certain amount.

- **all_change** returns shortest list first. Hence change becomes simple:

```
change amount = head (all_change amount)
```

- **all_change** returns all possible (decreasing sequences) of change for the given amount.

```
all_change :: Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
    c<-coins, amount>=c,
    cs<-all_change (amount - c) ]
```
all_change works by recursion from within a list comprehension. To make change for an amount we

1. Find the largest coin \( c \leq \text{amount} \):
   \[ c <- \text{coins}, amount > = c. \]

2. Find how much we now have left to make change for: \( \text{amount} - c \).

3. Compute all the ways to make change from the new amount: \( cs <- \text{all_change} (\text{amount} - c) \)

4. Combine \( c \) and \( cs \): \( c : cs \).
Example – Making Change...

If there is more than one coin \( c \leq \text{amount} \), then \( c \leftarrow \text{coins}, \text{amount} \geq c \) will produce all of them. Each such coin will then be combined with all possible ways to make change from \( \text{amount} - c \).

\textit{coins} returns the available coins in reverse order. Hence \textit{all_change} will try larger coins first, and return shorter lists first.

\begin{verbatim}
all_change :: Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
  c<-coins, amount>=c,
  cs<-all_change (amount - c) ]
\end{verbatim}
A list comprehension \([e \mid q]\) generates a list where all the elements have the form \(e\), and fulfill the requirements of the qualifier \(q\). \(q\) can be a generator \(x \leftarrow \text{list}\) in which case \(x\) takes on the values in \(\text{list}\) one at a time. Or, \(q\) can be a boolean expression that filters out unwanted values.
Homework

Show the lists generated by the following Haskell list expressions.

1. \([n \times n \mid n \leftarrow [1..10], \text{even } n]\)
2. \([7 \mid n \leftarrow [1..4]]\)
3. \([ (x, y) \mid x \leftarrow [1..3], y \leftarrow [4..7]]\)
4. \([ (m, n) \mid m \leftarrow [1..3], n \leftarrow [1..m]]\)
5. \([j \mid i \leftarrow [1,-1,2,-2], i > 0, j \leftarrow [1..i]]\)
6. \([a+b \mid (a, b) \leftarrow [(1,2), (3,4), (5,6)]]\)
Homework

Use a list comprehension to define a function `neglist xs` that computes the number of negative elements in a list `xs`.

**Template:**

```
neglist :: [Int] -> Int
neglist n = ...
```

**Examples:**

```
> neglist [1,2,3,4,5]
0
> neglist [1,-3,-4,3,4,-5]
3
```
Homework

Use a list comprehension to define a function `gensquares low high` that generates a list of squares of all the even numbers from a given lower limit `low` to an upper limit `high`.

**Template:**

```haskell
gensquares :: Int -> Int -> [Int]
gensquares low high = [ \[ \[ \[ \cdot \cdot \cdot ] | \cdot \cdot \cdot ] ]
```

**Examples:**

```haskell
> gensquares 2 5
[4, 16]
> gensquares 3 10
[16, 36, 64, 100]
```