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:
  \[ [ \text{expr} | \text{qualifier}, \text{qualifier}, \cdots ] \]
  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{qualifiers}.”

- The expression can be any valid Haskell expression.
- The qualifiers 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:
  pattern <- list_expr
  The pattern is often a simple variable. The list_expr is often an arithmetic sequence.

  \[ [n | n<-[1..5]] \Rightarrow [1,2,3,4,5] \]
  \[ [n*n | n<-[1..5]] \Rightarrow [1,4,9,16,25] \]
  \[ [(n,n*n) | n<-[1..3]] \Rightarrow [(1,1),(2,4),(3,9)] \]

Filter Qualifiers

- 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.

  \[ [n*n | n<-[1..9],even n] \Rightarrow [4,16,36,64] \]
  \[ [(n,n*n) | n<-[1..9],n<n*n] \Rightarrow [(2,4),(3,9)] \]
Local Definitions

- We can define a **local variable** within the list comprehension.
  
  Example:
  
  \[ [n \times n \mid \text{let } n = 2] \Rightarrow [4] \]

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

  **Haskell:**
  
  ```haskell
  [(i, j) \mid i \leftarrow [1..9], j \leftarrow [1..3]] \Rightarrow
  [(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) \mid i \leftarrow [1..3], j \leftarrow [i..4]] \Rightarrow
  [(1,1), (1,2), (1,3), (1,4),
  (2,2), (2,3), (2,4),
  (3,3), (3,4)]
  ```

  **Note:** \( n \times n \mid \text{let } n = [1..10], \text{even } n \) \Rightarrow \([4, 16, 36, 64, 100]\)

Example

- Define a function `doublePos xs` that doubles the positive elements in a list of integers.

  ```haskell
  doublePos :: [Int] -> [Int]
  doublePos xs = [2 * x \mid x \leftarrow xs, x > 0]
  ```

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

- Note that `xs` is a list-valued expression.
Example

- Define a function `spaces n` which returns a string of `n` spaces.

```haskell
spaces :: Int -> String
spaces n = [ ' ' | i <- [1..n] ]
```

Example: `spaces 10` ⇒ "

Pythagorean Triads:

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

```haskell
triads n = [(x,y,z) | x<-[1..n], y<-[1..n], z<-[1..n], x^2 + y^2 == z^2]
```

Example...
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:**

```plaintext
> 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]]
```

---

**all_change** works by recursion from within a list comprehension. To make change for an amount `amount` we:

1. Find the largest coin `c` ≤ `amount`: `<-coins,amount>=c`.
2. Find how much we now have left to make change for: `amount - c`.
3. Compute all the ways to make change from the new amount: `cs<-all_change (amount - c)`.

---

**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:

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

**Example – Making Change...**

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

```haskell
all_change :: Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
  c<-coins, amount>=c, c:cs<-all_change (amount - c) ]
```

---

- **If there is more than one coin `c` ≤ `amount`, then `<-coins,amount>=c` will produce all of them. Each such coin will then be combined with all possible ways to make change from `amount - c`.
- **coins** returns the available coins in reverse order. Hence `all_change` will try larger coins first, and return shorter lists first.

```haskell
all_change :: Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
  c<-coins, amount>=c, c:cs<-all_change (amount - c) ]
```
Summary

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

Homework

Use a list comprehension to define a function \(\text{neglist } xs\) that computes the number of negative elements in a list \(xs\).

\[\text{neglist} :: \{\text{Int}\} \rightarrow \text{Int}\]
\[\text{neglist } n = \cdots\]

Examples: __________________________
\[> \text{neglist } [1,2,3,4,5] \quad 0\]
\[> \text{neglist } [1,-3,-4,3,4,-5] \quad 3\]

Homework

Use a list comprehension to define a function \(\text{gensquares } \text{low} \text{ high}\) that generates a list of squares of all the even numbers from a given lower limit \(\text{low}\) to an upper limit \(\text{high}\).

\[\text{gensquares} :: \{\text{Int}\} \rightarrow \{\text{Int}\} \rightarrow \{\text{Int}\}\]
\[\text{gensquares } \text{low} \text{ high} = \{\cdots \mid \cdots \}\]

Examples: __________________________
\[> \text{gensquares } 2 \ 5 \quad [4, 16]\]
\[> \text{gensquares } 3 \ 10 \quad [16, 36, 64, 100]\]