CSc 372

Comparative Programming Languages

13 : Haskell — List Comprehension

Christian Collberg

collberg+372@gmail.com

Department of Computer Science

University of Arizona

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:

  ```
  [ expr | qualifier, qualifier, ··· ]
  ```

  In English, this reads:

  > "Generate a list where the elements are of the form expr, such that the elements fulfill the conditions in the 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]] ⇒ [1,2,3,4,5]

[n*n | n<-[1..5]] ⇒ [1,4,9,16,25]

[(n,n*n) | n<-[1..3]] ⇒ [(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] ⇒ [4,16,36,64]

[(n,n*n) | n<-[1..3],n<n*n] ⇒ [(2,4),(3,9)]
```

Local Definitions

- We can define a local variable within the list comprehension. Example:

  `[n*n | 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:

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

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]
```

Example

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

<u>In English:</u>

"Generate a list of elements of the form `2*x`, where the `x`:s are the positive elements from the list `xs`.

<u>In Haskell:</u>

```
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.

Example

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

Example:

```
> spaces 10
  "          "
```

Haskell:

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

- Note that the expression part of the comprehension is of type `Char`.
- Note that the generated values of `i` are never used.

Example

- Define a function `factors n` which returns a list of the integers that divide `n`. Omit the trivial factors `1` and `n`.

Examples:

```
factors 5 ⇒ []
factors 100 ⇒ [2,4,5,10,20,25,50]
```

In Haskell:

```
factors ::  Int -> [Int]
factors n = [i | i<-[2..n-1], n `mod` i == 0]
```

Example

Pythagorean Triads:

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

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

```
triads 5 ⇒ [(3,4,5),(4,3,5)]
```

Example...

- We can easily avoid generating duplicates:

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

triads' 11 ⇒ [(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:

```
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) ]
```

Example – Making Change. . .

- `all_change` works by recursion from within a list comprehension. To make change for an amount `amount` we
  1. Find the largest coin `c` $\leq$ `amount`: `c<-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)`
  4. Combine `c` and `cs`: `c:cs`.

Example – Making Change. . .

- If there is more than one coin `c` $\leq$ `amount`, then `c<-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.

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

Summary

- A list comprehension `[e|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<-list` in which case x takes on the values in list one at a time. Or, q can be a a boolean expression that filters out unwanted values.

Homework

- Show the lists generated by the following Haskell list expressions.

1. `[n*n | n<-[1..10],even n]`
2. `[7 | n<-[1..4]]`
3. `[ (x,y) | x<-[1..3], y<-[4..7]]`
4. `[ (m,n) | m<-[1..3], n<-[1..m]]`
5. `[j | i<-[1,-1,2,-2], i>0, j<-[1..i]]`
6. `[a+b | (a,b)<-[(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:

```
gensquares ::  Int -> Int -> [Int]
gensquares low high = [ ··· | ··· ]
```

Examples:

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