List Comprehension

Yan Huang

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$\{x^2 \mid x \in \{1...5\}\}$ $= \{1, 4, 9, 16, 25\}$

The set $\{1,4,9,16,25\}$ of all numbers x^2 such that x is an element of the set $\{1...5\}$.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

$[x\text{^}2 \mid x \leftarrow [1..5]]$

For $x = 1$ to 5

output x^2

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.

- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

For x from 1 to 3

For y from 4 to 5

output (x,y)

for y from 4 to 5

for x from 1 to 3

output (x, y).

- Changing the order of the generators changes the order of the elements in the final list:

[(4,1), (4,2), (4,3), (5,1), (5,2) (5,3)]

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

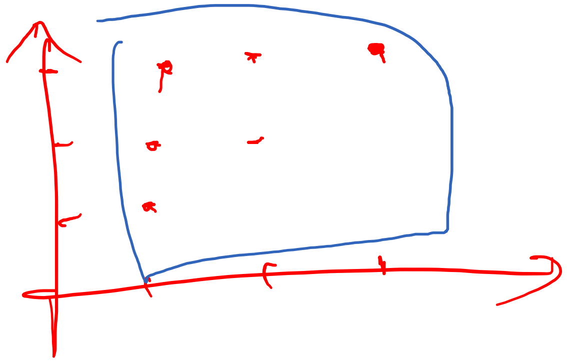
For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

x ← [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 introduced by earlier generators.

[(x,y) | x ← [1..3], y ← [x..3]]

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
of all pairs of numbers (x,y) such that x,y are
elements of the list [1..3] and y ≥ x.

Using a dependent generator we can define the library function that <u>concatenates</u> a list of lists:

concat :: [[a]] → [a]

concat xss = [x | xs ← xss, x ← xs]

Types: [a] [[a]] a [a]

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Using a dependent generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Guards

List comprehensions can use <u>guards</u> to restrict the values produced by earlier generators.

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

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

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

For example:

```
> factors 15

[1,3,5,15]
```

Using a guard we can define a function that maps a positive integer to its list of factors:

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

For example:

```
> factors 15

[1,3,5,15]
```

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime :: Int -> Bool
prime n = (factors n) == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

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

For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

primes :: Int → [Int]

primes n = [x | x ← [2..n], prime x]

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes   :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

pairs :: [a] → [(a, a)]
pairs xs = zip xs (tail xs)

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

pairs "abcde"

[('a','b'),('b','c'),('c','d'),
('d','e')]

Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

```
pairs    :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted :: [Int] -> Bool
sorted xs = and [x<=y | (x,y) <- pairs xs]
```

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted    :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y | (x,y) ← pairs xs]
```

<=

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all <u>positions</u> of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..], x == x']
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

```
"abc" :: String
```

Means ['a', 'b', 'c'] :: [Char].

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string:

```
count      :: Char → String → Int
count x xs =
   length [x’ | x’ ← xs, x == x’]
```

For example:

```
> count ’s’ "Mississippi"
4
```

Exercises

(1) A triple (x,y,z) of positive integers is called <u>pythagorean</u> if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500

[6,28,496]
```

(3) The scalar product of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i = 0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.