EECS 368 Programming Language Paradigms

David O. Johnson Fall 2022

Reminders

- Assignment 6 due: 11:59 PM, Monday, November 14
- Assignment 7 due: 11:59 PM, Wednesday, December 7

In-Class Problem Solution

• 28-(10-31) In-Class Problem Solution.pptx

Set Comprehensions

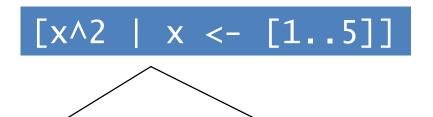
In discrete mathematics (EECS 210), the <u>comprehension</u> (or set builder) notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1...5\}\}$$

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 <u>lists</u> from old lists.



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.

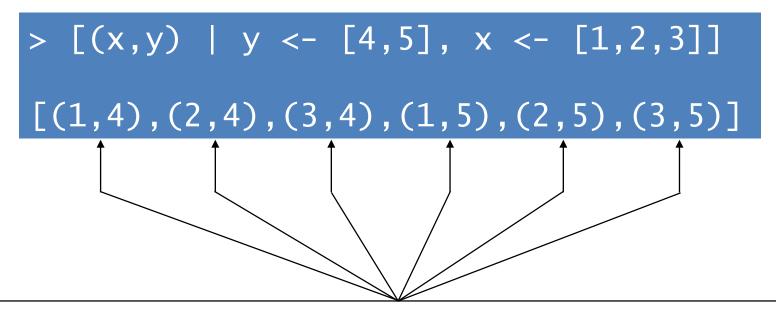
Multiple Generators

Comprehensions can have <u>multiple</u> generators, separated by commas. For example:

Changing the <u>order</u> of the generators changes the order of the elements in the final list:

Multiple Generators

Multiple generators are like <u>nested loops</u>, with later generators as more deeply nested loops whose variables change value more frequently. For example:



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 <u>depend</u> on the variables that are introduced by earlier generators.

$$[(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

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 $y \ge x$.

Dependent Generators

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]</pre>
```

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

List comprehensions can use guards to restrict the values produced by earlier generators.

$$[x \mid x < - [1..10], even x]$$

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 <u>factors</u>:

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

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

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

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

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

```
> prime 15
False
> prime 7
True
```

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

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]</pre>
```

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

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

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

The Zip Function – Example 1

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

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

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

The Zip Function – Example 2

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 \le y \mid (x,y) <- pairs xs]
```

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

The Zip Function – Example 3

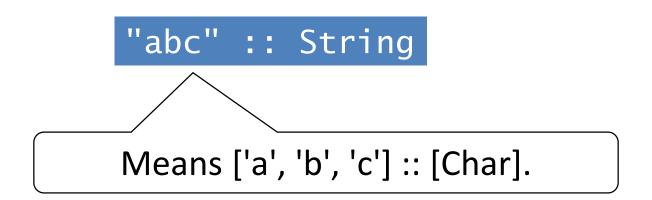
Using zip we can define a function that returns the list of all <u>positions</u> of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs =
   [i | (x',i) <- zip xs [0..], x == x']</pre>
```

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

String Comprehensions

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



String Comprehensions

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)]
```

String Comprehensions

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

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

```
> count 's' "Mississippi"
4
```

In-Class Problem

- 1. Using a list comprehension, give an expression that calculates the sum of the first one hundred integer squares: $1^2 + 2^2 + ... + 100^2$.
- 2. A triple (x,y,z) of positive integers is called pythagorean 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)]
```