CS571: Programming Languages

List Comprehension

Set

- Mathematically, the comprehension notation can be used to construct a new set from an old set

$$S = \{2*n \mid n \in \{1,2,3,4\}, n^2 > 3\}$$

S is the set of all 2*n where n is a member of the set {1,2,3,4} and n^2 is greater than 3

- n: variable which is a member of an input set.
- {1,2,3,4}: the input set.
- $n^2 > 3$: a Boolean expression used to restrict the ranges of the variable n
- 2*n: an output function producing members of the new net

List Comprehension

- [<expr> | <generators>, <filters>] -- denotes a finite or infinite list

```
Hugs>[2*n | n <- [1,2,3,4], n^2>3]
[4,6,8]
```

The list of all 2*n where n is a member of the list [1,2,3,4] and n^2 is greater than 3

- The symbol <- reads as "belongs to"
- n <- [1,2,3,4]: generator.
- When two or more generators are present, they are separated by commas .
- $n^2>3$: filter, a Boolean expressions used to restrict the ranges of the variable

Examples

```
Hugs>[(n,n) | n <- [2,4..10], n<10]

Hugs> [2*n| n <- [2,3,4,5], n>3]

Hugs>[(x,y) |x<-[1..3], y<-[3,7]]

Hugs>[(+) x y |x <-[1..3], y <-[3,7]]
```


Examples

```
Hugs>[(n,n) | n <- [2,4..10], n<10]
[(2,2),(4,4),(6,6),(8,8)]

Hugs> [2*n| n <- [2,3,4,5], n>3]

Hugs>[(x,y) |x<-[1..3], y<-[3,7]]

Hugs>[(+) x y |x <-[1..3], y <-[3,7]]
```

Examples

Hugs>[(n,n) | n <- [2,4..10], n<10]
[(2,2),(4,4),(6,6),(8,8)]

Hugs> [2*n| n <- [2,3,4,5], n>3]
[8,10]

Hugs>[(x,y) |x<-[1..3], y<-[3,7]]

Hugs>[(+) x y |x <-[1..3], y <-[3,7]]

Examples

Hugs>[(n,n) | n <- [2,4..10], n<10]
[(2,2),(4,4),(6,6),(8,8)]

Hugs> [2*n| n <- [2,3,4,5], n>3]
[8,10]

Hugs>[(x,y) |x<-[1..3], y<-[3,7]]
[(1,3),(1,7),(2,3),(2,7),(3,3),(3,7)]

Hugs>[(+) x y |x <-[1..3], y <-[3,7]]

Examples

Hugs>[(n,n) | n <- [2,4..10], n<10]
[(2,2),(4,4),(6,6),(8,8)]

Hugs> [2*n| n <- [2,3,4,5], n>3]
[8,10]

Hugs>[(x,y) |x<-[1..3], y<-[3,7]]
[(1,3),(1,7),(2,3),(2,7),(3,3),(3,7)]

Hugs>[(+) x y |x <-[1..3], y <-[3,7]]
[4,8,5,9,6,10]

More Examples

Hugs>[2*x | x <- [0..], x^2 > 3]

More Examples

Hugs>[2*x | x <- [0..], x^2 > 3]

[4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100,102,104,106,108,110,112,114,116,118,120,122,124,126,128,130,132,134,136,138,140,142,144,146,148,150,152,154,156,158,160,162,164,166,168,170,172,174,176,178,180,182,184,186,188,190,192,194,196,198,200,202 ……

Example: doubleAll

- doubleAll: double every element of an integer list
  doubleAll [2,3] = [4,6]
  doubleAll [4,2,3] = [8,4,6]

List comprehension

Without using list comprehension

Example: doubleAll

- doubleAll: double every element of an integer list
  doubleAll [2,3] = [4,6]
  doubleAll [4,2,3] = [8,4,6]

List comprehension

```
doubleAll xs = [2*x| x <- xs]
```

Without using list comprehension

Example: doubleAll

- doubleAll: double every element of an integer list
  doubleAll [2,3] = [4,6]
  doubleAll [4,2,3] = [8,4,6]

List comprehension

```
doubleAll xs = [2*x| x <- xs]
```

Without using list comprehension

```
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```


Example: single

- Write a list comprehension single x that changes a list into a list of lists by making each element into a singleton list.
  e.g. > single [1,2,3,4]
  [[1],[2],[3],[4]]

List comprehension

Without using list comprehension

Example: single

- Write a list comprehension single x that changes a list into a list of lists by making each element into a singleton list.
  e.g. > single [1,2,3,4]
  [[1],[2],[3],[4]]

List comprehension

```
single list = [[x]| x <- list]
```

Without using list comprehension

Example: single

- Write a list comprehension single x that changes a list into a list of lists by making each element into a singleton list.
  e.g. > single [1,2,3,4]
  [[1],[2],[3],[4]]

List comprehension

```
single list = [[x]| x <- list]
```

Without using list comprehension

```
single [] = []
single (x:xs) = [x]: single(xs)
```


Example: sqroot

- Write a simple list comprehension that finds the square root of an integer n if the number is an exact square.
  E.g. > sqroot 4
  [2]
  > sqroot 5
  []

Example: sqroot

- Write a simple list comprehension that finds the square root of an integer n if the number is an exact square.

E.g. > sqroot 4
[2]
> sqroot 5
[]

```
sqroot n = [x | x<-[1..n],x*x==n]
```


Example: sqroot

- Write a simple list comprehension that finds the square root of an integer n if the number is an exact square.

E.g. > sqroot 4
[2]

Solution1:

```
sqroot n = sqroot1 [1..n] n

sqroot1 [] n = []
sqroot1 (x:xs) n
  |x*x == n = [x]
  |otherwise = sqroot1 xs n
```


Example: sqroot

- Write a simple list comprehension that finds the square root of an integer n if the number is an exact square.

E.g. > sqroot 4
[2]

Solution 2:

```
sqroot n = sqroot1 n 1

sqroot1 n k
    |n == k = []
    |k*k == n = [k]
    |otherwise = sqroot1 n (k+1)
```


Example: Library Function concat

- Concatenate a list of lists

e.g. > concat [[1,2],[3,4],[5]]
[1,2,3,4,5]

Example: Library Function concat

- Concatenate a list of lists

e.g. > concat [[1,2],[3,4],[5]]
[1,2,3,4,5]

```
concat list = [x | sublist <- list, x <- sublist]
```


Example: Library Function concat

- Concatenate a list of lists

e.g. > concat [[1,2],[3,4],[5]]
[1,2,3,4,5]

```
concat list = [x | sublist <- list, x <- sublist]
```

Without using list comprehension

```
concat [] = []
concat (x:xs) = x  ++ concat xs
```

AddLists

- addLists list1 list2 : add every element of list1 to every element of list2.

```
addLists :: [Int] -> [Int] -> [Int]
Main> addLists [1,2] [3,4]
[4,5,5,6]
```


AddLists

- addLists list1 list2 : add every element of list1 to every element of list2.

```
addLists :: [Int] -> [Int] -> [Int]
Main> addLists [1,2] [3,4]
[4,5,5,6]

addLists list1 list2 = [x+y| x <- list1, y <- list2]
```


Quicksort (Haskell)

- Pick an element, called a pivot, from the list.
- Reorder the list so that all elements which are less or equal to the pivot is in one sub-list A and elements greater than the pivot is in another sub-list B.
- Recursively sort the sub-list A and the sub-list B.

Quicksort (Haskell)

- Pick an element, called a pivot, from the list.
- Reorder the list so that all elements which are less or equal to the pivot is in one sub-list A and elements greater than the pivot is in another sub-list B.
- Recursively sort the sub-list A and the sub-list B.

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs)
        = quicksort [y| y<-xs, y<=x]
    ++ [x]
    ++ quicksort [y| y<- xs, y>x]
```


User-Defined Types

User Defined Types: Enumerated

- User Defined Types: type constructors
  - Enumerated type: enumerating the elements of the type

```
data Color = Red | Blue | Black
```

Red, Blue and Black are data constructors of the type Color

User Defined Types: Product

```
data Shape =  Circle Float              -- Radius
            | Rectangle Float Float  -- length & width

area :: Shape -> Float
area (Circle r) = 3.14 * r*r
area (Rectangle h w) = h*w

Main> area (Circle 3.1)
30.1754

Main> area (Rectangle 3 4)
12.0
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

```
2
2+3
(3-1)+3
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

```
2
2+3
(3-1)+3
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

```
2       Lit 2
2+3
(3-1)+3
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

```
2       Lit 2
2+3     Add (Lit 2) (Lit 3)
(3-1)+3
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

```
2       Lit 2
2+3     Add (Lit 2) (Lit 3)
(3-1)+3 Add (Sub (Lit 3) (Lit 1)) (Lit 3)
```


User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- Evaluate an expression

```
Main> eval (Add (Lit 2) (Lit 3))
5
```

User Defined Types: Recursive

- Expression: an arithmetic expression is either a literal integer, like 347, or is given by combining two expressions using an arithmetic operator e.g. +, -

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- Evaluate an expression

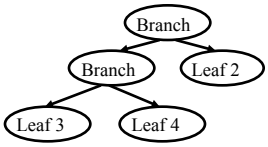
```
Main> eval (Add (Lit 2) (Lit 3))
5

eval :: Expr -> Int
eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```


Example: Tree

```
data Tree  = Leaf Int | Branch Tree Tree
```

Branch (Branch(Leaf 3)(Leaf 4)) (Leaf 2)



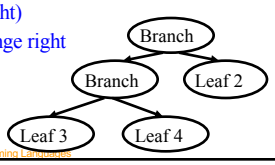
Example: Tree

```
data Tree  = Leaf Int | Branch Tree Tree
```

fringe: returns a list of all the elements in the leaves of a tree from left to right.

```
> fringe (Branch(Branch(Leaf 3)(Leaf 4))(Leaf 2))
[3,4,2]

fringe :: Tree  -> [Int]
fringe (Leaf x) = [x]
fringe (Branch left right)
   = fringe left ++ fringe right
```

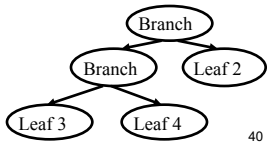


Example: height

```
data Tree = Leaf Int
          | Branch Tree Tree
```

- Write a haskell function height tr to compute the height of the tree tr.

eg. > height (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3))
3



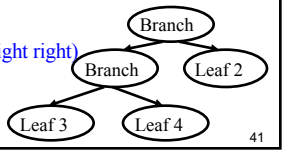
Example: height

```
data Tree = Leaf Int
          | Branch Tree Tree
```

- Write a haskell function height tr to compute the height of the tree tr.

eg. > height (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3))
3

```
height (Leaf _) = 1
height (Branch left right) =
    1 + max (height left) (height right)
```

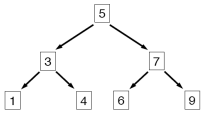


Example: collectnode

- Consider the binary tree defined below:

```
data Tree = Leaf Int | Node Tree Int Tree
```

E.g. Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9)) specifies the tree:



Write a function **collectnode n bt** that collects all nodes of a binary tree bt that are greater than n.

E.g. collectnode 5 (Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9)))
[6,7,9]

Example: collectnode

```
data Tree = Leaf Int | Node Tree Int Tree

collectnode :: Int -> Tree -> [Int]
collectnode n (Leaf y)
      | y> n = [y]
      | otherwise = []
collectnode n (Node tr1 n1 tr2)
      | n1 > n = (collectnode n tr1) ++ [n1] ++ (collectnode n tr2)
      | otherwise = (collectnode n tr1) ++ (collectnode n tr2)
```