
csci54 – discrete math & functional programming
tuples and lists

Recap

- ▶ Write a function `cap'` that not only caps the upper limit at 100, but additionally evaluates to 0 if `n` is less than or equal to 0.
- ▶ Write a function `pow` that takes two parameters `n` and `k` and returns `n` to the `k`th power. (assume that `k` is guaranteed to be a non-negative integer. do not use the `**` operator)

```
cap' n =  
  if n > 100  
  then 100  
  else  
    if n < 0  
    then 0  
    else n
```

```
cap n =  
  if n > 100  
  then 100  
  else n  
  
cap' n =  
  if n < 0  
  then 0  
  else (cap n)
```

```
pow n k =  
  if k == 0  
  then 1  
  else n * (pow n (k-1))
```

maxInt

- ▶ write a function `maxInt` that takes a list of integers and returns the value of the largest element. you may assume the list is not empty.

```
maxInt [x] = x
maxInt (x:xs) = max x (maxInt xs)
```



Lists in Haskell

- ▶ Homogeneous (all same type)
- ▶ square brackets with element separated by commas
- ▶ building lists
 - ▶ square brackets with values separated by commas

```
ghci> aList = [1, 10, -3, 5]
```

- ▶ cons

```
ghci> aList2 = 2 : [1, 10, -3, 5]
```

- ▶ concatenation

```
ghci> aList3 = aList ++ aList2
```



Lists in Haskell continued

- ▶ functions on lists

- ▶ head, tail
- ▶ init, last
- ▶ take, drop
- ▶ length, null
- ▶ reverse
- ▶ ...

```
aList = [2, 1, 10, -3, 5]
```

- ▶ ``elem`` vs `elem`

- ▶ infix vs. prefix

```
elem 1 [2, 1, 10, -3, 5]
```

```
-->true
```

```
1 `elem` [2, 1, 10, -3, 5]
```

```
-->true
```

- ▶ same with arithmetic functions: `div`, `mod`

- ▶ `div`: round down
- ▶ `mod`: integer mod (goes with `div`)

```
(Haskell also has quot, rem, which  
behave differently than div/mod with  
negative numbers)
```

Practice problems

- ▶ what does this function do?

```
numList n =  
  if n <= 0  
  then []  
  else  
    n : (numList (n-1))
```



Practice problems

- ▶ (on week01-ps) numList n evaluates to a list of integers from n down to 1

```
numList n =  
  if n <= 0  
  then []  
  else  
    n : (numList (n-1))
```

- ▶ numList 3 →
 - 3 : numList 2 →
 - 3 : (2 : numList 1) →
 - 3 : (2 : (1 : numList 0)) →
 - 3 : (2 : (1 : []))) == [3, 2, 1]



Practice problems

- ▶ (on week01-ps) numList n evaluates to a list of integers from n down to 1

```
numList n =  
  if n <= 0  
  then []  
  else  
    n : (numList (n-1))
```

- ▶ Write a function oddList where oddList n evaluates to a list of odd integers from n down to 1. If $n < 1$ the function should return an empty list.
 - ▶ Write a function oddList' where oddList' evaluates to a list of odd integers from 1 up to, but possibly not including, n. If $n < 1$ the function should return an empty list. Do not use the reverse function.
-



-
- In this example, will aList and bList be the same at the end?

```
aList = [2, 1, 10, -3, 5]  
bList = 2:aList  
aList = 2:aList
```



List comprehensions (and ranges)

- ▶ A way to build up lists:

```
[ x*2 | x <- [1..3] ]
```

- ▶ Note use of ranges in Haskell

```
[ 1,3..10 ]  
[ 10,9..1 ]
```

```
[ 1,4..  
[ 47.. ]
```

- ▶ Can add more to list comprehensions:

```
[ x*y | x <- [1..3], y <- [6,4,2] ]
```

```
[ x*y | y <- [6,4,2], x <- [1..3] ]
```



More on list comprehensions

- ▶ Can add predicates:

```
[ x*y | x <- [1..3], y <- [1..3], x > y]
```

- ▶ Can use any expression:

```
[ if x*y > 3 then "BIG" else "SMALL" | x <- [1..3], y <- [1..3]]
```

```
[ (x,y) | x <- ['a'..'c'], y <- ["rat","ox","tiger"]]
```

- ▶ a tuple does not need to be homogeneous; cannot append or concatenate, so must know number of elements from start



Practice problems

- ▶ Write a function `oddList` where `oddList n` evaluates to a list of odd integers from `n` down to 1. If `n < 1` the function should return an empty list.
- ▶ Write a function `oddList'` where `oddList'` evaluates to a list of odd integers from 1 up to, but possibly not including, `n`. If `n < 1` the function should return an empty list
- ▶ Rewrite `oddList` and `oddList'` using list comprehensions
- ▶ What do these evaluate to?

```
[ if x*y > 3 then [1] else [2] | x <- [1..3], y <- [1..3] ]  
[ (x,y,z) | x <- [1..3], y <- [1..3], z <- [1..3], x < y, y < z ]  
[ (x,y,z) | z <- [1..3], y <- [1..3], x <- [1..3], x < y, y < z ]
```