Andrew Krall
09-14-2018

List Functions Continued

reverse: Reverses a list. How it works is that it takes a list and returns a list with the elements in the opposite order.

We don't need to write efficient Haskell code, but it's good to think about efficiency.

"Accumulator Passing" (Harrison's term) Style is more efficient. Idea is that you accumulate the right answer in the accumulator and return it when it's all accumulated.

HW 2 asks you to write some simple problems with accumulator passing style.

drop: Drops the number of elements from the beginning of a list, with the number and list being given as parameters.

```
drop :: Int -> [a] -> [a] -- Takes an Int and a list and returns a list of things.
drop 0 ls = ls
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

maximum: On HW 2! The idea with maximum is that it takes a list of stuff (non-negative Ints) in order and returns the largest value.

```
maximum :: [Int] -> Int
maximum [] = 0
maximum (x:xs) = if x > maximum xs then x else maximum xs
```

→

Andrew Kvalı
09-14-2018

Can also do

```
maximum (x:xs) = if x > maxxs then x else maxxs
                 where maxxs = maximum xs
```

so that it's only called once.

Ranges

You can denote [1..20] to show a list from 1-20 (like in Bash if statements).

nub: Removes duplicates from a list.

repeat: Takes a list and makes it an infinite list.

Haskell has infinite lists through "lazy evaluation", will learn more about this later.

```
repeat :: a -> [a]
repeat x = x : repeat x

take 10 <list>   (Shows first 10 of the list)

cycle :: [a] -> [a]
cycle l = l ++ cycle [a]
```

Set Comprehensions: Way of denoting what a set is.

This is a set comprehension.

$$S = \{2 * x \mid x \in Nat, x \leq 10\}$$

→

CS4450 Lecture 12 Notes, Page 3

Andrew Kroll
09-14-201

List Comprehension: Way of defining a list in Haskell.

List comprehension examples are listed in the slides. You can make them part of functions.

Tuples: Built-in type constructors for ordered pairs, ordered triples, etc. E.g. ("Wow", 'a') is an ordered pair.

The Prelude-defined functions for pairs are polymorphic (can be used with any type).

Zippers

zip: zip [1,2,3,4,5] [5,5,5,5,5]

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

What type is zip? [a] → [b] → [(a,b)]

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []  -- If one list empty, return empty list
zip _ [] = []  -- Likewise, do same here if other list empty
zip (a:as) (b:bs) = [(a,b)] ++ zip as bs
↳ Better way: = (a,b) : zip as bs
```

When you have an append (++), look back and see if you could have used a cons (:)!