Data Structures: Lists

Winter 2015-2016

Table of Contents

Append

```
append :: [a] -> [a] -> [a]

append []  ls  = ls
append (x:xs)  ls  = x : (append xs ls)
```

- Parens are needed because function application is higher precedence than the list constructor `:`.
- Note that the parens are not needed on the right hand side of the equation. There is no other function application there to confuse issues.
- But Haskell has append builtin: infix operator `++`

```
Hugs> [1,2,3] ++ [4,5,6]
 [1,2,3,4,5,6]
```

- On the LHS, `:` is a pattern, on the RHS, it is a constructor.

Length

- Example: length

```
length :: [a] -> Int

length [] = 0
length (x:xs) = 1 + (length xs)
```

- But Haskell has it already defined.

reverse

- Example: reverse: return the list in the reverse order

```
reverse :: [a] -> [a]

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

- This version is not very efficient: $O(n^2)$

reverse

- A better version

```
myreverse      ::  [a] -> [a]
myreverse xs   =  rev xs  []
  where rev []  ys          =  ys
        rev (x:xs) ys       =  rev xs (x:ys)
```

- Tail recursive
- The accumulator is like a stack; last-on, first off

Head and Tail

- It is convenient to define functions that return the head and the tail of a list:

```
head :: [a] -> a
head (x:xs)  = x

tail  :: [a] -> [a]
tail  (x:xs) = xs
```

- When writing list recursion, use `x:xs`. Use **head**, **tail** when you're not doing list recursion.

Init

- The function **init** returns everything but the last element

```
Hugs> init [2,3,4]
[2,3]
```

- Implementation

```
init :: [a] -> [a]
init [x] = []
init (x:xs) = x : init xs
```

- Notes:
  - Two non-exclusive patterns
  - Correctness depends on the order of the equations!

Last

- The function **last** returns the last element in a list

```
Hugs> last [2,3,4]
4
```

- Implementation

```
last :: [a] -> a
last (x :[])  = x
last (x:xs)   = last xs
```

take

- Example: take: return the first *n* elements

```
take :: Int -> [a] -> [a]

take   0      xs    = []
take (n+1)    []    = []
take (n+1) (x:xs)  = x : take n xs
```

- Haskell has it already defined
- Note the "magic" patterns!
  - **take** (n+1) [] = []
  - A positive integer is allowed to match (n+1) in Haskell
  - A special case to make functions a little cleaner

```
Hugs> take 3 [1,2,3,4,5,6]
[1,2,3]
```

drop

- Example: drop: return all elements after the *n*th

```
drop :: Int -> [a] -> [a]

drop 0 xs = xs
drop (n+1) [] = []
drop (n+1) (x:xs) = drop n xs
```

- Haskell has it already defined

```
Hugs> drop 3 [1,2,3,4,5,6]
 [4,5,6]
```