

CS339: Abstractions and Paradigms for Programming

Lists in Haskell

Manas Thakur
CSE, IIT Bombay



Autumn 2024

Recall the Common Words *program*

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                  sortRuns . countRuns . sortWords .
                  words . map toLower
```

- We are yet to define showRun, sortRuns, countRuns and sortWords.
- showRun is easy:

```
showRun :: (Int, Word) -> String
showRun (n,w) = w ++ ": " ++
                show n ++ "\n"
```

- The remaining will allow us learn some more Haskell.



Pattern Matching in Haskell

- Lists in Haskell are denoted using square brackets: `[1,2,3]`
- Empty list (our favorite base case) is just empty brackets: `[]`
- *cons* in Haskell is denoted using :

`x:xs`, where `x` is car and `xs` is cdr

- Check if a list is empty:

```
null :: [a] -> Bool
null []      = True
null (x:xs)  = False
```

OR

```
null :: [a] -> Bool
null [] = True
null _  = False
```

cons is non-strict in both arguments.

An example is better than 10 definitions

- We can define *map* as follows:

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

- What about *filter*:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

- Notice the indentation below the *if*.



Lists: The omnipresent data structure in APP



Enumerations

- `[1..10]`
 - `[1,2,3,4,5,6,7,8,9,10]`
- `[1..]`
 - `[1,2,3,4 {C-c}`
- `[0,2..11]`
 - `[0,2,4,6,8,10]`
- `[1,3..]`
 - `[1,3,5,7 {C-c}`

- `['a' .. 'z']`
 - `[a,b,c,d,e,f,g,h,i,j,k,
l,m,n,o,p,q,r,s,t,u,v,w
,x,y,z]`
- Smart enough, but not so much!
 - `[20..1] vs [20,19..1]`
 - `[1,2,4,8..100]` won't work



Some cool things with enumerations

- Get first 24 multiples of 13:
 - `[13, 26 .. 24*13]`
- Cooler:
 - `take 24 [13, 26 ..]`
- Why does it work?
 - Lazy evaluation!

- `cycle [1, 2, 3]`
 - Hang!
- `take 10 (cycle [1, 2, 3])`
 - `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]`
- `take 11 (cycle "LOL ")`
 - `"LOL LOL LOL"`



List comprehensions

- Describe the set of first 10 even natural numbers:
 - Math: $S = \{x*2 \mid x \in \mathbb{N}, x \leq 10\}$
 - Haskell: `[x*2 | x <- [1..10]]`
- Elements between 1 and 10 which when doubled are greater than 12 but less than 80 when multiplied by 3:
 - `[x | x <- [1..10], x*2 > 12, x*3 < 80]`

Told you they are close!



Cooler things with list comprehensions

- Prime numbers between 1 and 100:

- `[x | x <- [1..100],
isPrime x]`

- First 100 prime numbers:

- `take 100 [x | x <- [1..],
isPrime x]`

- All iteration vectors for summing two nxn matrices:

- `[(i,j) | i <- [1..n],
j <- [1..n]]`

- map:

- `map f xs = [f x | x <- xs]`

- filter:

- `filter p xs = [x | x <- xs,
p x]`
Order order

- concat:

- `concat xss = [x | xs <- xss,
x <- xs]`

Notice the nested *loops* here!



Back to Common Words

- Here is a possible definition of countRuns:

```
countRuns :: [Word] -> [(Int,Word)]
countRuns [] = []
countRuns (w:ws) = (1+length us,w) : countRuns vs
                  where (us,vs) = span (==w) ws
```

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span p [] = ([], [])
span p (x:xs) = if p x then (x:ys,zs)
                else ([],x:xs)
                where (ys,zs) = span p xs
```



Guards

- Now we're left with `sortWords` and `sortRuns`, but let's first define a **merge sort**:

```
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
           where (ys,zs) = half xs

half xs = (take n xs, drop n xs)
           where n = length xs `div` 2

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```


It's an easy finisher now

```
sortWords :: [Word] -> [Word]
sortWords = sort

sortRuns :: [(Int,Word)] -> [(Int,Word)]
sortRuns = reverse . sort
```

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                  sortRuns . countRuns . sortWords .
                  words . map toLower
```



Summary of case constructs in Haskell

Next class:

Typed Programming with Haskell

- If-then-else
- Guards
- Pattern matching
- **Case analysis**
- There are even more :p

```
head xs = case xs of
            [] -> error "Empty list"
            (x:_) -> x
```



Syntactic sugar at its best!