COMP105 Lecture 19

Higher Order Programming Examples

Outline

Today

- Higher order programming examples
  - Mark averages
  - Voting systems

Relevant book chapters

- Programming In Haskell Chapter 7
- Learn You a Haskell Chapter 6

Mark averages

We have a file of **student marks**

- For assignment 1, 2, 3, and the class test

```
aaaa      70   65   67   60
bbbb      55   60   55   65
cccc      40   40   40   40
dddd      80   60   75   60
cccc      0     0    0  100
```

Mark averages

We want to produce a file of **mark averages**

```
aaaa 65.5
bbbb 58.75
cccc 40.0
dddd 68.75
cccc 25.0
```

Reading files in Haskell

We can read a file using **readFile**

- This is an IO function
- We will study this in more detail later on

```
ghci> readFile "marks.csv"
"aaaa     70  65  67  60\nbbbb     55  60  55...
```

The `'\n'` character is the **newline** character

lines

The function **lines** gives us a list of lines

```
ghci> lines "line 1\nline 2\nline 3\n"
["line 1","line 2","line 3"]
```

```
ghci> file <- readFile "marks.csv"

ghci> lines file
["aaaa      70  65  67  60",
 "bbbb      55  60  55  65", ...
```

unlines

The **unlines** function does the opposite

```
ghci> unlines ["line 1", "line 2", "line 3"]
"line 1\nline 2\nline 3\n"
```

```
ghci> unlines . lines $ file
"aaaa    70  65  67  60\nbbbb    55  60  55  65
```

Parsing the file

Using **words** and **lines** we can parse the file

```
ghci> let parsed = map words . lines $ file

ghci> parsed
[["aaaa","70","65","67","60"],
 ["bbbb","55","60","55","65"],
 ["cccc","40","40","40","40"],
 ["dddd","80","60","75","60"],
 ["cccc","0","0","0","100"]]
```

Getting the averages

```
average :: [String] -> Float
average [student, a1, a2, a3, ct] =
          (read a1 + read a2 + read a3 + read ct) / 4
```

```
ghci> let averages = map average parsed
ghci> averages
[65.5,58.75,40.0,68.75,25.0]
```

Getting the student names

```
name :: [String] -> String
name [student, _, _, _, _] = student
```

```
ghci> let names = map name parsed
ghci> names
["aaaa","bbbb","cccc","dddd","cccc"]
```

Creating the report

```
report_line :: String -> Float -> String
report_line student average =
        student ++ " " ++ show average
```

```
ghci> let zipped = zipWith report_line names averages
ghci> zipped
["aaaa 65.5",
 "bbbb 58.75",
 "cccc 40.0",
 "dddd 68.75",
 "cccc 25.0"]
```

Writing the output file

```
ghci> unlines zipped
"aaaa 65.5\nbbbb 58.75\ncccc 40.0\n..."

ghci> writeFile "report.csv" (unlines zipped)
```

All in one function

```
report file =
    let
        parsed   = map words . lines $ file
        students = map name parsed
        averages = map average parsed
        zipped   = zipWith report_line students averages
    in
        unlines zipped
```

Voting: first past the post

In a first past the post election, whoever gets the **most votes** wins

```
ghci> winner ["red", "blue", "red", "red", "green"]
"red"
```

Exercise: how would you implement `winner`?

Getting the candidates

First we need to figure out who the candidates are

```
uniq [] = []
uniq (x:xs) = x : uniq (filter (/=x) xs)
```

```
ghci> uniq ["red", "red", "blue", "green", "red", "blue"]
["red","blue","green"]
```

Counting the votes

This function counts the number of votes for a particular candidate

```
count x list = length (filter (==x) list)
```

```
ghci> count "red" ["red", "blue", "red", "red", "blue"]
3
```

Vote totals

```
totals votes =
    let
        candidates = uniq votes
        f = (\ c -> (count c votes, c))
    in
        map f candidates
```

```
ghci> totals ["red", "blue", "red", "red", "blue"]
[(3,"red"),(2,"blue")]
```

Finding the winner

Recall: tuples are ordered **lexicographically**

```
ghci> max (3, "red") (2, "blue")
(3,"red")
```

```
ghci> maximum [(3, "red"), (2, "blue"), (4, "green")]
(4,"green")
```

Finding the winner

```
winner votes = (snd . maximum . totals) votes
```

```
ghci> winner ["red", "blue", "red", "red", "green"]
"red"
```

Alternative vote

In the **alternative vote** system, voters rank the candidates

- In each round, the candidate with the least number of first preference votes is eliminated
- The winner is the last candidate left once all others have been eliminated

```
ghci> let votes = [["red", "blue", "green"],
                   ["blue", "green"],
                   ["green", "red"],
                   ["blue", "red"],
                   ["red"]]

ghci> av_winner votes
"red"
```

Getting the first choice votes

```
first_choice votes = map head votes
```

```
ghci> let votes = [["red", "blue", "green"],
                   ["blue", "green"],
                   ["green", "red"],
                   ["blue", "red"],
                   ["red"]]
```

```
ghci> first_choice votes
["red","blue","green","blue","red"]
```

Ranking the candidates

```
import Data.List

rank votes = sort . totals . first_choice $ votes
```

```
ghci> let votes = [["red", "blue", "green"],
                   ["blue", "green"],
                   ["green", "red"],
                   ["blue", "red"],
                   ["red"]]
```

```
ghci> rank votes
[(1,"green"),(2,"blue"),(2,"red")]
```

Removing a losing candidate

```
remove_cand c votes =
let
    rm_votes = map (filter (/=c)) votes
    rm_empty = filter (/=[]) rm_votes
in
    rm_empty
```

```
ghci> remove_cand "green" votes
[["red","blue"],["blue"],["red"],["blue","red"],["red"]]
```

```
ghci> remove_cand "red" votes
[["blue","green"],["blue","green"],["green"],["blue"]]
```

Putting it all together

```
av_winner votes =
    let
        ranked = rank_candidates votes
        first = head ranked
    in
        if length ranked == 1
        then first
        else av_winner (remove_cand first votes)
```

```
ghci> av_winner votes
"red"
```

Summary

- Higher order programming examples

Next time: Custom types