

Reading

18. Monday 28 Feb: Sethi, Section 3.1.

Exercises

Due date: Tuesday 1 March. Please leave your paper in the Taylor Hall homework box (it's located between 2.132 and 2.136) **by 4pm**.

These exercises require you to write Haskell programs. You should run each program on Hugs, and include some output, demonstrating the function, immediately following the Haskell code.

1. [5] In yet another version of occurrences (HW3, Exercise 4), look for opportunities to use `foldr`, `map`, and `filter`.
2. [15] Redefine the functions of HW 3 Exercise 6 using `foldr`. Demonstrate your definitions on a few examples using Hugs.
 - a. Define `conc` so that `conc xss` returns the concatenation of the lists. For example,
`conc ["the", " ", "hungry", " ", "i"] ~ "the hungry i"`
 - b. Define `remDups` so that `remDups xs` returns the elements of `xs` with adjacent duplicates removed. For example,
`remDups [3,4,4,5,2,2,2,9,8,2,3] ~ [3,4,5,2,9,8,2,3]`
 - c. Define `average` so that if `ns` is a non-empty list of `Ints`, `average ns` is their average. A simple definition of average would be
`> average :: [Int] -> Float`
`> average ns = fromInt (sum ns) / fromInt (length ns)`
but this makes two passes over `ns`. Define a more efficient implementation of `average` which makes a single pass, using the function
`sumAndLength :: [Int] -> (Int,Int)`
This function satisfies the equation
`sumAndLength ns = (sum ns, length ns)`
but produces its result in a single pass over `ns`.
3. [5] Redefine the function `insertSort` of HW 3 Exercise 7 using `foldr`.
4. [15] Define an infinite list of factorials two ways.
 - a. Define it as simply as possible, using the function `fac :: Int -> Integer` in a list comprehension.
 - b. Define a more efficient version of the factorial list, using memoization to avoid redundant computations.
 - c. Run both definitions on some examples, using Hugs with the statistics option turned on (the Hugs command is `:set +s`). Is the difference in efficiency detectable?
5. [10] Mathematically, a function can be defined as a set of ordered pairs. For functions with small domains, that can be a practical implementation. Use a list of tuples to implement the function that maps the names of states to the names of their capital cities, where names are implemented by `Strings` (you don't need to include all 50 states; a dozen or so will do). Define a function which, applied to a state name, returns the corresponding city name. If there is no match, the function should return an empty `String`.
6. [15] Define a function that computes the value of e^x using the series expansion
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$
In addition to `facList` (from Exercise 4) and `powersOf` (from the lecture notes, Slide 96) you may find use for the Prelude functions `zipWith` and `takeWhile`. Your solution should discard all terms of the series smaller than 10^{-6} , and should avoid redundant computation.

7. [10] Define `getLine` using `getChar`.

To avoid collisions with names defined in the Prelude, you can make your script import the Prelude explicitly, *hiding* the names you're redefining. For example,

```
import Prelude hiding (getLine)
```

8. [10] Define a function `doAll :: [IO ()] -> IO ()` which takes a list of IO actions and returns an IO action which performs the elements of the list in sequence.

Use `doAll` to define a function `printList` so that `printList xs` is an action which displays the elements of `xs` on successive lines.

9. [15] Define an I/O action `readSort` which reads integers on successive lines until it encounters a line which is not an integer. After reading the last integer, it should output a sorted sequence of the integers it has read, one per line. The action should prompt for its inputs and explain its output.

10. [25] Define a parser `numeral` for C/C++ integer numerals, using the following syntax:

```
Numeral ::= Octal | Decimal | Hexadecimal
```

```
Octal ::= 0 { OctalDigit }
```

```
Decimal ::= PosDigit { DecimalDigit }
```

```
Hexadecimal ::= 0X { HexDigit }
```

```
OctalDigit ::= 0 | 1 | ... | 7
```

```
PosDigit ::= 1 | 2 | ... | 9
```

```
DecimalDigit ::= 0 | PosDigit
```

```
HexDigit ::= 0 | 1 | ... | 9 | 'A' | 'B' | ... | 'F'
```

Your parser should deliver the Integer whose value the numeral represents.

11. [10] Generalize `getInt` to make an IO action `get :: Parser a -> IO a` which takes a parser as its argument and applies it to a line of input. Then use `get` and `numeral` to define `getNumeral`.

CS 345 Richards

Assignment 4

Authors:

1. _____

2. _____

3. _____

4. _____