

Reading

- 10. Wednesday 9 Feb: Davie: Sections 2.8, 2.10, 2.11, 3.11
Gentle Intro: Section 2.4.1, 2.4.2
- 11. Friday 11 Feb: Davie: Sections 2.9, 3.4, 3.10.1, 3.10.2
- 12. Monday 14 Feb: Davie: Section 3.10.3
- 13. Wednesday 16 Feb: Gentle Intro: Sections 3.4, 7.0.
- 14. Friday 18 Feb: Gentle Intro,: Sections 7.1, 7.2, 7.5.

Exercises

Due date: Tuesday 15 Feb. Please leave your paper in the Taylor Hall homework box (it's located in the breezeway between 2.132 and 2.136) **by 4pm.**

Whenever an exercise calls for a Haskell definition, you should include in your answer a few examples demonstrating your definition using Hugs98. The demonstration should be included in your paper by copy-and-paste from the Hugs worksheet.

1. [15] Define a function `facTable` so that `facTable k` produces a nicely formatted table of factorials from $0!$ to $k!$. For example, `facTable 5` would produce

n	n!
0	1
1	1
2	2
3	6
4	24
5	120

Use the type `Integer`, rather than `Int`, for the factorial values.

Note that the table's columns are right-justified. Consider defining a function `rjustify` so that `rjustify w s` produces a string containing `s` with enough leading blanks to make a string of length `w`.

Demonstrate your definition using Hugs for $k = 30$.

2. [15] Design a function `dollar` which produces a `String`, representing an amount of money, in a form suitable for printing on a check. Define `dollar` so that `dollar w c` is a `String` of length `w` (or more, if necessary); its first character, '\$', should be followed by enough '*' characters so that the amount is right-justified in the `String`.

The amount `c` is a positive `Integer` representing *cents*, but `dollar` should display it in the usual decimal-dollar format, with a decimal point followed by two digits. In computing the dollars and cents portions of the result, do **not** convert the amount to `Float` or `Double`, because this can introduce imprecision (banks hate that!. Instead, use `div` and `mod`.

For example,

```
dollar 10 250 ~ "$*****2.50"
dollar 10 2   ~ "$*****0.02"
dollar 1 250  ~ "$2.50"
```

Hint: This exercise is quite easy if you make good use of local definitions in a **where**-clause to modularize your function definition (divide and conquer!).

3. [15] Define a Haskell function `fromThenTo` such that

`fromThenTo a b c = [a,b..c]` for all values `a`, `b`, and `c` of type `Int`

without using the `..` notation. Experiment with Hugs to find out how `[a,b..c]` behaves for various relationships among `a`, `b`, and `c` (i.e., $b < a < c$, $a < b < c$, $a = b = c$, etc.), and use what you discover to define your function. Remember to declare the function's type.

Demonstrate your definitions on a few examples using Hugs.

4. [10] Define a function `occurrences` so that `occurrences xs` returns a list containing one copy of each element of `xs` paired with the number of times it occurs. For example,
- ```
occurrences [5,2,3,2,4,3,2,5] ~ [(5,2), (2,3), (3,2), (4,1)]
```
- Don't restrict yourself to a single function definition— consider using auxiliary functions to break the task into simpler subtasks. For this exercise, define `occurrences` to work for `Int` lists.
- Consider two different approaches, using list comprehensions and explicit recursion, respectively. Demonstrate your definitions on a few examples using Hugs.
- In one version of `occurrences`, use list comprehensions as much as possible.
  - In another version of `occurrences`, use explicit recursion only.
5. [10] Using your `occurrences` function from Exercise 4, define a function `uniques` so that `uniques ns` is a list containing only the elements of `ns` that occur exactly once in `ns`. For example,
- ```
uniques [1,2,3,4,5,4,3] ~ [1,2,5]
```
6. [30] Define the following functions recursively. Demonstrate your definitions on a few examples using Hugs.
- `conc`, so that, given a list of lists `xss`, `conc xss` returns the concatenation of the lists. For example,

```
conc ["the", " ", "hungry", " ", "i"] ~ "the hungry i"
```

 - `remDups`, so that if `ns` is a list of `Ints`, `remDups ns` returns the elements of `ns` 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]
```

 - `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`.
7. [10] Define a function `insert` so that if `as` is a sorted list of `Ints`, `insert a as` returns a sorted list in which `a` appears in its proper place. For example,
- ```
> insert 5 [1,3,6,7] ~ [1,3,5,6,7]
> insert 6 [1,3,6,7] ~ [1,3,6,6,7]
```
- Then define `insertSort`, which uses `insert` to sort a list of `Ints`. Demonstrate your definitions on a few examples using Hugs.

CS 345 Richards

Assignment 3

Authors:

1. _____

2. _____

3. _____

4. _____