CSE 341, Spring 2018, Assignment 3
Haskell Warmup
Due: Monday April 16, 10:00pm

12 points total (2 points each)

You can use up to 2 late days for this assignment.

Some of these questions involve defining the same function in different styles, to get experience with the
different possibilities in Haskell.

For each top-level Haskell function you define, include a type declaration. For example, your cone_volume
function for Question 1 should start with:

```
cone_volume :: Double->Double->Double
```

Please turn in one file called HaskellWarmup.hs that contains all your functions, type declarations, and unit
tests.

1. Write a function cone_volume that takes a radius and height of a cone and returns its volume. Remem-
ber to include the type declaration. (Double is a built-in Haskell type representing a double-precision
floating point number.) If you write this function without a type declaration and let Haskell infer the
type, it will actually come up with a more general type – but we’re going to ease into Haskell’s type
system and just declare the function to use Doubles. Also use the defined Haskell constant pi.

2. Write a ascending function to test whether a list of integers is in strict ascending order. For
example, ascending [1,2,3], ascending [10], and ascending [] should all return True, while
ascending [2,3,1] and ascending [2,2] should both return False.

3. Write a function squares that takes a list of integers, and returns a list of the squares of those integers.
For example, squares [1,2,10] should evaluate to [1,4,100], while squares [] should evaluate to []
Also try your function on an infinite list, for example squares [1..] or squares [1,3..].
For full credit your function should be written in the pointfree style, i.e. the definition should start
squares = ... (no named argument). Hint: what is the type of(^2) ? What does it do?

4. Write a function parallel_resistors that calculates the total resistance of a number of resistors
connected in parallel. The formula for computing this is

\[ \frac{1}{r_1} + \ldots + \frac{1}{r_n} \]

where \( r_1, \ldots, r_n \) are the resistances of each resistor. For example, parallel_resistors [10.0, 10.0],
representing two 10.0 Ohm resistors in parallel, should return 5.0. Hint: if you make good use of
functions in the Haskell prelude (e.g. map and recip) this is a one-line definition. This version should
not be pointfree.

Don’t worry about the edge cases of zero Ohm resistors, or no resistors (i.e. an empty list as an
argument to parallel_resistors). However, after you’ve written your function, try it on these cases
and see what happens. Explain why you get the results that you do in a comment.

5. Write a function pointfree_parallel_resistors that is a pointfree version of the
parallel_resistors function from Question 4. Hint: if you’re stuck, think about how to
build this up in easy steps. What’s an expression that will compute this list?

\[ \frac{1}{r_1}, \ldots, \frac{1}{r_n} \]
Once you’ve got that, what expression will compute this?

\[
\frac{1}{r_1} + \ldots + \frac{1}{r_n}
\]

6. A palindrome is a sentence or phrase that is the same forwards and backwards, ignoring spaces, punctuation and other special characters, and upper vs. lower case. Some palindromes are “Madam, I’m Adam”, “Yreka Bakery”, and “Doc, note, I dissent, a fast never prevents a fatness. I diet on cod.” We’ll also consider digits (but not special characters like /) as part of the sentence or phrase — so that 01/02/2010 also counts as a palindrome. Write a Haskell function `palindrome` that takes a string as an argument and that returns True if the string is a palindrome and otherwise False. Hint: make use of the Haskell library for this problem; you shouldn’t need to write a recursive function at all for your solution. In particular, consider using functions such as `map`, `filter`, `reverse`, and assorted functions in the `Data.Char` module. If you use this module, include an `import Data.Char` statement in your program.

Use the HUnit unit testing package to write test cases for each function. Include a test for an ordinary case, and also tests for edge cases. Also include a `run` function that runs all of your tests. (See the `UnitTestExample.hs` file linked from the Haskell page in the 341 website.)

For example, for the palindrome problem, include tests for a string that is a palindrome, a string that isn’t a palindrome, the empty string, and a date palindrome:

```haskell
p1 = TestCase (assertBool "banana palindrome" (palindrome "Yo! Banana Boy!"))
p2 = TestCase (assertBool "carrot palindrome" (not (palindrome "Yo! Carrot Girl!")))
p3 = TestCase (assertBool "empty palindrome" (palindrome ""))
p4 = TestCase (assertBool "date palindrome" (palindrome "01/02/2010"))
```

To test the `squares` function evaluated on an infinite list, use the `take` function from the Haskell Prelude to get the first several values and check them. (Otherwise `assertEqual` isn’t going to be happy if you just give it infinite lists.) For example:

```haskell
s1 = TestCase (assertEqual "infinite squares" [1,9,25,49] (take 4 (squares [1,3 ..])))
```

For the tests involving floating-point numbers, as usual you should test for equality within an epsilon, rather than exact equality (which would often fail due to roundoff errors). If you would like, you can copy the following definition of a convenience function into your own program:

```haskell
{- test whether a number is within epsilon of to another (for unit tests on
Doubles, to accomodate floating point roundoff errors). Note that this doesn’t
work for testing numbers that should be exactly 0 -- for that you should specify
your own test with an appropriate epsilon -}
is_close x y = abs (x-y) < abs x * epsilon
    where epsilon = 1.0e-6
```

As is often the case, there will probably be significantly more code for the tests than for the actual functions!

Your program should be tastefully commented (i.e. put in a comment before each function definition saying what it does). Style counts! In particular, think about where you can use pattern matching and higher order functions to good effect to simplify your program; and avoid unnecessary repeated computations.