CSE 341, Autumn 2008, Assignment
Haskell Warmup
Due: Monday September 29, 10:00pm

12 points total (2 points each for Questions 1–3; 3 points each for Questions 4 and 5)

You can use up to 2 late days for this assignment. (The reason for this is so that we can post sample solutions within a reasonable time — if we wait for 6 days that's getting long.)

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

```
sphere_volume:: Double->Double
```

1. Write a function `sphere_volume` that takes a Double representing the radius of a sphere, and that returns the Double that is the volume of the sphere. Remember 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 take a Double and return a Double. Hint: use the built-in constant `pi`.

2. Write a function `cone_volume` that takes two numbers (both Double) representing the height and radius of the base of a cone, and that returns the volume of the cone. (To be precise, since in Haskell all functions are curried, `cone_volume` doesn't actually take two parameters; instead, it takes one parameter and returns a function that takes another parameter, which finally returns the volume.) Again, Remember to include the type declaration.

3. Write a recursive function `cubes` that takes a list of integers (type `Int`), and returns a list of the cubes of those integers. For example, `cubes [1,3,10]` should evaluate to `[1,27,1000]`, while `cubes []` should evaluate to `[]`. Also try your function on an infinite list, for example `cubes [1..]` or `cubes [1,3..]`.

4. Write another version of the `cubes` function, called `map_cubes`, that uses the built-in `map` function in Haskell. `map_cubes` should not be recursive. Don't define a named helper function – use an anonymous function.

5. Write a function `duplicate` that takes a list `s` and an integer `n`, and returns a list with n copies of s concatenated together. The list `s` can contain any other type. If n is negative, give an error message. Here are some examples:

```
duplicate [10, 20, 30] 2   =>   [10, 20, 30, 10, 20, 30]
duplicate [10, 20, 30] 0   =>   []
duplicate [True] 4         =>   [True, True, True, True]
duplicate "No!" 5          =>   "No!No!No!No!No!"
duplicate "No!" (-5)       =>   Exception: negative argument to duplicate
```

   Hints: `duplicate` has a polymorphic type:

```
duplicate :: [a] -> Int -> [a]
```

   See the `rec_factorial` function in the lecture notes for an example of checking for a bad input parameter value.

Turnin: Turn in your Haskell program and a script showing it running on some well-chosen test cases. Your program should be tastefully commented (i.e. put in a comment before each function definition saying what it does).