CSc 372 — Comparative Programming Languages
Spring 2026 (McCann)

https://cs.arizona.edu/classes/cs372/spring26/

Program #2: Haskell

Due Date: March 23rd, 2026, at the beginning of class

Overview: In this class, we study specific languages in some detail, but not all detail. Our purpose is to study the design of the languages. In the case of Haskell, we are focusing on its functional characteristics. The best way to understand those features is to write programs that use them. Before you can do that, you also need to be able to use the basics of the language. That is why this assignment is a collection of programming exercises, ranging from the straight–forward to the more involved. Between the lecture material and this assignment, we hope that you will become comfortable enough with Haskell to continue to explore additional features of the language on your own, should you choose to do so.

General Requirements: The following requirements apply to all parts of this assignment:

1. Write a complete, well–documented Haskell code for each numbered part. The required file name is given with each part. The example programs on the class web page are all written as complete programs; follow their lead when writing your programs.
2. Each part has at least one required function. For these, we have provided the function definition's first line (its type signature); use it. If you want to write helper functions, great! Haskell programs are often large collections of helper functions.
3. The use of existing Haskell functions (from the Prelude or any other module) that trivialize a problem is prohibited. For example, if we ask you to write a function that adds the elements of a list, you could not just call `sum` to do it.
4. **AI Tool Usage:** You may use AI tools (LLMs, Generative AIs, IDE AI plug–ins, etc.) to help you write code on the programming assignments in this class. However, we *strongly* recommend that you use it to help you only when you get stuck, because you will be asked to write code on the exams. For that reason, write the code yourself, as much as possible, to help you learn the languages covered in this class.

Assignment: For each numbered task below, write complete, well–documented Haskell code (named as stated) that accomplishes it.

1. **Mean of a List of Integers**.

   - Program file name: `mean.hs`
   - Function signature: `meanIntList :: [Int] -> Float`
   - Sample Function Invocation:

   ```
   ghci> meanIntList [3,4,8]
   5.0
   ```

   An easy starter: Write `meanIntList`, which averages the integers in the given list and returns their mean as a real number. Produce a meaningful error message when the given list is empty.

   Hints: `fromIntegral` is your friend, and the '`error` *String*' function is good for displaying error messages.

   Reminder: Using a built–in averaging function is not allowed, but using a built–in summing function, for example, to help you write `meanIntList` is allowed.

(Continued...)

2. **Greatest Common Divisor**.

   - Program file name: `gcd.hs`
   - Function signature: `ourGCD :: Int -> Int -> Int`
   - Sample Function Invocation:

```
ghci> ourGCD 780 612
12
```

   GCD is a classic recursive function. Given two integers, their GCD is:

```
gcd(x, y) =  gcd(y, x % y)
```

   assuming both arguments are positive. If one argument is zero, the GCD is equal to the other argument. If both are zero, or if either is negative, display appropriate error messages. (The Prelude has a function named `gcd`; of course, per the directions at the top of this assignment, you may **NOT** use it in this program.)

3. **Hailstones**.

   - Program file name: `hailstones.hs`
   - Function signature: `hailstones :: Int -> [Int]`
   - Sample Function Invocation:

```
ghci> hailstones 7
[7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
```

   Yes, this is the same piecewise function you worked with in Program #1: Starting with a positive integer, halve it if it's even, otherwise triple it and add one. Stop when one is reached.

   As shown, we want the hailstone sequence returned as a list of integers, starting with the given value and ending with one. (That's right; no fancy output formatting.) When the argument is one, the result should be the list containing just one. Display a helpful error message when an argument less than one is provided.

4. **Lower Right Triangular Matrix**

   - Program file name: `lrtm.hs`
   - Function signature: `lrtm :: Int -> [[Int]]`
   - Sample Function Invocations:

```
ghci> lrtm 1
[[1]]
ghci> lrtm 4
[[0,0,0,4],[0,0,3,4],[0,2,3,4],[1,2,3,4]]
```

   Consider an n–by–n matrix, and imagine a line drawn diagonally through the matrix from the upper right to the lower left. A matrix that contains non–zero data only on or below that line is a lower right triangular matrix.

   The function `lrtm` accepts a positive integer n and returns a two–dimensional list of the rows, in top–row to bottom–row order, of the content of the lower right triangular matrix that contains, on row r (the first row is row 1, per mathematical convention), the last r values of the sequence $1..n$ in the last r columns, and zeroes in the remaining locations.

(Continued...)

That description is a bit dense. For example, consider this 4×4 matrix, which is what the result of `lrtm 4` would look like if presented as a 2D matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 3 & 4 \\ 0 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Consider the third row from the top ($r = 3$). It contains, in the last three columns, the last three values of the sequence 1..4, and a zero in the remaining column.

Notes: We are not asking you to create a 'pretty–printed' version of the matrix; see the sample invocations for the desired return values. If a non–positive integer is supplied, display an appropriate error message.

5. **Vowel Histogram**.

   - Program file name: `vowels.hs`
   - Function signatures: `vowelListList :: String -> [String]`

     `vowelHistogram :: [String] -> String`
   - Sample Function Invocations:

```
ghci> vowelListList "Ate\na horse?\nAaa! Ick, ick, ick!\n"
["aaaaa","ee","iii","o",""]
ghci> vowelHistogram ["aaaaa","ee","iii","o",""]
"a: *****\ne: **\ni: ***\no: *\nu: "
```

   I/O is a bit of a problem for functional languages, due to side effects. Still, no language would be practical without it. For this program, you will need write `main` to read the content of a text file named "vowels.txt" and store the file's stream of characters into a String. (`main` will also display the final histogram.) You may assume that the file "vowels.txt" will exist and is readable.

   `vowelListList` will accept the string containing the file content and return a list of five strings, one per lower-case English vowel ('a', 'e', 'i', 'o', and 'u', in that order) with one occurrence of the appropriate vowel for each occurrence of that vowel in the argument, regardless of case. If a vowel does not appear in the text, its string is the empty string. (See the 'u' string in the example above.)

   `vowelHistogram` will accept such a list of Strings and will produce a String that, when printed with `putStr` or `putStrLn`, produces a simple horizontal histogram of the vowels. For example, this is the result `Main` should display from the example shown above:

```
$ ./vowels
a: *****
e: **
i: ***
o: *
u:
```

   Note that each line begins with a vowel, a colon, and a space (even the 'u' line has a space after the colon).

(Continued...)

6. **Linear Insertion Sort**.

   - Program file name: `insertionsort.hs`
   - Function signatures: `insertInOrder :: Int -> [Int] -> [Int]`
     `insertionSort :: [Int] -> [Int]`
   - Sample Function Invocations:

```
ghci> insertInOrder 4 [1,6,9,12]
[1,4,6,9,12]
ghci> insertionSort [12,6,1,9,4,15,2]
[1,2,4,6,9,12,15]
```

   In the first program, you wrote a version of Insertion Sort called Binary Insertion Sort. For this part of this program, you'll be writing 'regular' Insertion Sort, formally known as Linear Insertion Sort because the assumption is that the insertion locations in the growing sorted list are located using a sequential search, rather than binary search.

   For this implementation, you can write `insertInOrder` to find the correct insertion location by searching the list forward or backward; it's your choice. It is given an integer to be inserted and an ascending–ordered list of integers. `insertInOrder` should be able to handle inserting a value into an empty list.

   `insertionSort` accepts a list of integers and uses the Linear Insertion Sort algorithm to sort the list into ascending order. Of course, we expect that you will write `insertionSort` to call `insertInOrder` as a helper. If asked to sort an empty list, `insertionSort` is to return an empty list.

7. **Digitize**.

   - Program file name: `digitize.hs`
   - Function signature: `digitize :: String -> String`
   - Sample Function Invocation:

```
ghci> digitize "12:56"
"    _   _  _ \n  | _|._|_ |_ \n  ||_ . _||_|"
```

   No surprise: This part was inspired by the digital clock program from Program #1. This task is different: Given a string containing digits and/or colons, return a String consisting of three lines of space, underscore, 'pipe', and/or period characters that, when printed with `putStr` or `putStrLn`, will display the characters as if displayed by a digital clock.

   Each digit's digital representation is nine characters in size, in a 3x3 block. Thus, the digit '1' consists of seven spaces and two pipe characters, with the pipe characters on the right. The colon is just a column of three characters – a space at the top, and two periods. There are no additional spaces separating the digital representations.

   For example, the result string shown above, if printed, will look like this (the squashed 'u' represents a space character, the crude box surrounds it so that the trailing 'u's are displayed by LaTeX's `fancyvrb` package, and the 'clean' version to the right is provided to help you see the digits more plainly):

```
+-------------+
|␣␣␣␣_␣␣␣_␣␣␣_␣␣␣␣|     _   _  _
|␣␣|␣_|.|_␣|_␣|   | _|.|_ |_
|␣␣||_␣.␣_||_||   ||_ . _||_|
+-------------+
```

   If the argument contains characters other than digits and colons, use `error` to display a descriptive error message. If the argument is the empty string, return the string `"\n\n"`. Note that `digitize` is to be able to handle an input string of any length, so long as the content is digits or colons.

(Continued...)

Data: The input expectations for the programs are given with the program descriptions, above.

Output: Output details for each program are stated in the Assignment section, above. Please ask (preferably via a public post on Piazza) if you need additional details.

Hand In: You are required to submit your completed program files (your `.hs` files) using the `turnin` facility on lectura. The submission folder is `cs372p2`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because we will be grading your program on lectura, it needs to run on lectura, so be sure to test it on lectura. Submit all files as–is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

Additional Hints, Reminders, and Requirements:

- Only Vowel Histogram requires the creation of a complete, executable program. You may find it useful to create testing programs for the other functions, but providing testing programs is not part of the assignment. Remember, sharing testing code is OK! You're welcome to post testing code on Piazza if you'd care to do so.
- There is a list of Haskell resources on the class web page, and other suggestions (ghci, Hoogle, etc.) were mentioned in class. Working through statement syntax, execution errors, etc., are chores that we expect you to try hard to do on your own, as you would have to if you were learning a language on your own. But, if you get really stuck, the TAs and I are here to help.
- We realize that we haven't yet demonstrated all of the features of Haskell that you will need to complete all of these programs. You can either wait for those features to be covered, or you can explore them on your own. We strongly encourage the latter!
- I said this above, but it bears repeating: Functional programs are often little more than collections of functions that, called in the proper order, form a program. The secret to writing programs in a functional language is to divide the big tasks into many smaller tasks, each with a function to perform them. Many of those smaller tasks will be common ones that your program can use existing Haskell functions to perform. Start each program by designing the sequence of function calls that will solve the problem, then create and test each function, starting with the simplest, lowest–level functions, building slowly to a completed program.
- **Start Early!** Haskell has a way of driving novice functional programmers a little crazy, particularly programmers used to imperative and/or object–oriented languages. The earlier you start, the less crazed you will become. We don't expect that any of these programs will be *conceptually* difficult for you, but they will probably take some time to complete, due to your inexperience with Haskell and its functional paradigm.