Real World Haskell Exercises

Exercise #1

1. Enter the following expressions into ghci. What are their types?
   - 5 + 8
   - 3 * 5 + 8
   - 2 + 4
   - (+) 2 4
   - sqrt 16
   - succ 6
   - succ 7
   - pred 9
   - pred 8
   - sin (pi / 2)
   - truncate pi
   - round 3.5
   - round 3.4
   - floor 3.7
   - ceiling 3.3
2. From ghci, type `:?` to print some help. Define a variable, such as `let x = 1`, and then type :show bindings. What do you see?
3. The words function breaks a string up into a list of words. Modify the `WC.hs` example in order to count the number of words in a file.
4. Modify the `WC.hs` example again, in order to print the number of characters in a file.

Exercise #2

1. Haskell provides a standard function, `last :: [a] -> a`, that returns the last element of a list. From reading the type alone, what are the possible valid behaviors (omitting crashes and infinite loops) that this function could have? What are a few things that this function clearly cannot do?
2. Write a function, `lastButOne`, that returns the element before the last.
3. Load your lastButOne function into ghci and try it out on lists of different lengths. What happens when you pass it a list that's too short?

Exercise #3

1. Write the converse of fromList for the List type: a function that takes a `List a` and generates a `[a]`.

2. Define a tree type that has only one constructor, like our Java example. Instead of the Empty constructor, use the Maybe type to refer to a node's children.

Exercise #4

1. Write a function that computes the number of elements in a list. To test it, ensure that it gives the same answers as the standard length function.

2. Add a type signature for your function to your source file. To test it, load the source file into ghci again.

3. Write a function that computes the mean of a list, i.e., the sum of all elements in the list divided by its length. (You may need to use the fromIntegral function to convert the length of the list from an integer into a floating-point number.)

4. Turn a list into a palindrome; i.e., it should read the same both backward and forward. For example, given the list `[1,2,3]`, your function should return `[1,2,3,3,2,1]`.

5. Write a function that determines whether its input list is a palindrome.

6. Create a function that sorts a list of lists based on the length of each sublist. (You may want to look at the sortBy function from the Data.List module.)

7. Define a function that joins a list of lists together using a separator value:

   ```
   -- file: ch03/Intersperse.hs
   intersperse :: a -> [[a]] -> [a]
   ```

   The separator should appear between elements of the list, but it should not follow the last element. Your function should behave as follows:

   ```
   ghci> :load Intersperse
   [1 of 1] Compiling Main
    ( Intersperse.hs, interpreted )
   Ok, modules loaded: Main.
   ghci> intersperse ',' []
   ""
   ghci> intersperse ',' ["foo"]
   "foo"
   ghci> intersperse ',' ["foo","bar","baz","quux"]
   "foo,bar,baz,quux"
   ```

8. Using the binary tree type that we defined earlier in this chapter, write a function that will determine the height of the tree. The height is the largest number of hops from the root to an Empty. For example, the tree Empty has height zero; Node "x" Empty Empty has height one; Node "x" Empty (Node "y" Empty Empty) has height two; and so on.

9. Consider three two-dimensional points, a, b, and c. If we look at the angle formed by the line segment from a to b and the line segment from b to c, it turns left, turns right, or forms a straight line. Define a Direction data type that lets you represent these possibilities.

10. Write a function that calculates the turn made by three two-dimensional points and returns a Direction.

11. Define a function that takes a list of two-dimensional points and computes the direction of each successive triple. Given a list of points `[a,b,c,d,e]`, it should begin by computing the turn made by `[a,b,c]`, then the turn made by `[b,c,d]`, then `[c,d,e]`. Your function should return a list of Direction.

12. Using the code from the preceding three exercises, implement Graham's scan al gorithm for the convex hull of a set of 2D points. You can find good description of what a convex hull (http://en.wikipedia.org/wiki/Convex_hull) is, and how the Graham scan algorithm (http://en.wikipedia.org/wiki/Graham_scan) should work, on Wikipedia (http://en.wikipedia.org/).

Exercise #5

1. Write your own "safe" definitions of the standard partial list functions, but make sure they never fail. As a hint, you might want to consider using the following types:

```
-- file:
 ch04/ch04.exercises.hs
safeHead
 :: [a] -> Maybe a
safeTail
 :: [a] -> Maybe [a]
safeLast
 :: [a] -> Maybe a
safeInit
 :: [a] -> Maybe [a]
```

2. Write a function splitWith that acts similarly to words but takes a predicate and a list of any type, and then splits its input list on every element for which the predicate returns False:

```
-- file: ch04/ch04.exercises.hs
splitWith :: (a -> Bool) -> [a] -> [[a]]
```

3. Using the command framework from the earlier section "A Simple Command-Line Framework" on page 71, write a program that prints the first word of each line of its input.

4. Write a program that transposes the text in a file. For instance, it should convert "hello\nworld\n" to "hw\neo\nlr\nll\nod\n".

Exercise #6

1. Use a fold (choosing the appropriate fold will make your code much simpler) to rewrite and improve upon the asInt function from the earlier section "Explicit Re- cursion" on page 85.

```
-- file: ch04/ch04.exercises.hs
asInt_fold :: String -> Int
```

2. Your function should behave as follows:

```
ghci> asInt_fold "101"
101
ghci> asInt_fold "-31337"
-31337
ghci> asInt_fold "1798"
1798
```

3. Extend your function to handle the following kinds of exceptional conditions by

```
calling error:
ghci> asInt_fold ""
0
ghci> asInt_fold "-"
0
ghci> asInt_fold "-3"
-3
ghci> asInt_fold "2.7"
*** Exception: Char.digitToInt: not a digit '.'
ghci> asInt_fold "314159265358979323846"
564616105916946374
```

4. The `asInt_fold` function uses error, so its callers cannot handle errors. Rewrite the function to fix this problem:

```
-- file: ch04/ch04.exercises.hs
type ErrorMessage = String
asInt_either :: String -> Ei
ghci> asInt_either "33"
Right 33
ghci> asInt_either "foo"
Left "non-digit 'o'"
```

5. The Prelude function concat concatenates a list of lists into a single list and has the following type:

```
-- file: ch04/ch04.exercises.hs
concat :: [[a]] -> [a]
```

6. Write your own definition of concat using foldr.

7. Write your own definition of the standard takeWhile function, first using explicit recursion, and then foldr.

8. The Data.List module defines a function, groupBy, which has the following type:

```
-- file: ch04/ch04.exercises.hs
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

9. Use ghci to load the Data.List module and figure out what groupBy does, then write your own implementation using a fold.

10. How many of the following Prelude functions can you rewrite using list folds?

- any
- cycle
- words
- unlines For those functions where you can use either foldl' or foldr, which is more ap- propriate in each case?

Exercise #7

1. Our current pretty printer is spartan so that it will fit within our space constraints, but there are a number of useful improvements we can make. Write a function, fill, with the following type signature:

```
-- file: ch05/Prettify.hs
fill :: Int -> Doc -> Doc
```

It should add spaces to a document until it is the given number of columns wide. If it is already wider than this value, it should not add any spaces.

2. Our pretty printer does not take nesting into account. Whenever we open paren- theses, braces, or brackets, any lines that follow should be indented so that they are aligned with the opening character until a matching closing character is en- countered. Add support for nesting, with a controllable amount of indentation:

```
-- file: ch05/Prettify.hs
fill :: Int -> Doc -> Doc
```

Exercise #8

1. Load the Control.Arrow module into ghci and find out what the second function does.
2. What is the type of `(,)`? When you use it in ghci, what does it do? What about `(,,)`?

Exercise #9

1. Use ghci to explore what happens if you pass a malformed pattern, such as `[,` to `globToRegex`. Write a small function that calls globToRegex, and pass it a malformed pattern. What happens?
2. While filesystems on Unix are usually case-sensitive (e.g., "G" vs. "g") in filenames, Windows filesystems are not. Add a parameter to the globToRegex and matchesGlob functions that allows control over case sensitive matching.

Exercise #10

1. Although we've gone to some lengths to write a portable namesMatching function, the function uses our case sensitive globToRegex function. Find a way to modify namesMatching to be case-sensitive on Unix, and case insensitive on Windows, without modifying its type signature. (Hint: consider reading the documentation for System.FilePath to look for a variable that tells us whether we're running on a Unix-like system or on Windows.)
2. If you're on a Unix-like system, look through the documentation for the System.Posix.Files module, and see if you can find a replacement for the doesNameExist function.
3. The `*` wild card matches names only within a single directory. Many shells have an extended wild card syntax, `**`, that matches names recursively in all directories. For example, `**.c` would mean "match a name ending

in `.c` in this directory or any subdirectory at any depth". Implement matching on `**` wild cards.

Exercise #11

1. Write a version of globToRegex that uses the type signature shown earlier.
2. Modify the type signature of namesMatching so that it encodes the possibility of a bad pattern, and make it use your rewritten globToRegex function.

Exercise #12

1. What should you pass to traverse to traverse a directory tree in reverse alphabetic order?
2. Using id as a control function, traverse id performs a preorder traversal of a tree: it returns a parent directory before its children. Write a control function that makes traverse perform a postorder traversal, in which it returns children before their parent.
3. Take the predicates and combinators from "Gluing Predicates Together" on page 224 and make them work with our new Info type.
4. Write a wrapper for traverse that lets you control traversal using one predicate and filter results using another.

Exercise #13

1. Modify `foldTree` to allow the caller to change the order of traversal of entries in a directory.
2. The `foldTree` function performs preorder traversal. Modify it to allow the caller to determine the order of traversal.
3. Write a combinator library that makes it possible to express the kinds of iterators that `foldTree` accepts. Does it make the iterators you write any more succinct?

Exercise #14

1. Although the file-finding code we described in this chapter is a good vehicle for learning, it's not ideal for real systems programming tasks, because Haskell's portable I/O libraries don't expose enough information to let us write interesting and complicated queries. Port the code from this chapter to your platform's native API, either `System.Posix` or `System.Win32`.
2. Add the ability to find out who owns a directory entry to your code. Make this information available to predicates.

Exercise #15

1. Write a parser for "plain" PGM files.
2. In our description of "raw" PGM files, we omitted a small detail. If the "maximum gray" value in the header is less than 256, each pixel is represented by a single byte. However, it can range up to 65,535, in which case, each pixel will be represented by 2 bytes, in big-endian order (most significant byte first). Rewrite the raw PGM parser to accommodate both the single and double-byte pixel formats.
3. Extend your parser so that it can identify a raw or plain PGM file, and then parse the appropriate file type.

Exercise #15

Let's briefly explore the suitability of tuples as stand-ins for arrays: 1. Write a function that takes two arguments: a four-element tuple and an integer. With an integer argument of zero, it should return the leftmost element of the tuple. With an argument of one, it should return the next element. And so on. What restrictions do you have to put on the types of the arguments in order to write a function that typechecks correctly? 2. Write a similar function that takes a six-tuple as its first argument. 3. Try refactoring the two functions to share any common code you can identify. How much shared code are you able to find?

Exercise #16

1. Using QuickCheck, write a test for an action in the MonadHandle monad, in order to see if it tries to write to a file handle that is not open. Try it out on safeHello.
2. Write an action that tries to write to a file handle that it has closed. Does your test catch this bug?
3. In a form-encoded string, the same key may appear several times, with or without values, e.g., `key&key=1&key=2`. What type might you use to represent the values associated with a key in this sort of string? Write a parser that correctly captures all of the information.

Exercise #17

1. Our HTTP request parser is too simple to be useful in real deployments. It is miss- ing vital functionality and is not resistant to even the most basic denial-of-service attacks. Make the parser honor the Content-Length field properly, if it is present.
2. A popular denial-of-service attack against naive web servers is simply to send un- reasonably long headers. A single header might contain 10s or

100s of megabytes of garbage text, causing a server to run out of memory. Restructure the header parser so that it will fail if any line is longer than 4,096 characters. It must fail immediately when this occurs; it cannot wait until the end of a line eventually shows up.

3. Add the ability to honor the Transfer-Encoding: chunked header if it is present. See section 3.6.1 of RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec3 .html#sec3.6.1) for details.
4. Another popular attack is to open a connection and either leave it idle or send data extremely slowly. Write a wrapper in the IO monad that will invoke the parser. Use the System.Timeout module to close the connection if the parser does not complete within 30 seconds.

Exercise #18

1. Modify the App type synonym to swap the order of ReaderT and WriterT. What effect does this have on the runApp execution function?
2. Add the WriterT transformer to the App monad transformer stack. Modify runApp to work with this new setup.
3. Rewrite the constrainedCount function to record results using the WriterT trans- former in your new App stack.

Exercise #19

1. Write a many parser, with type `Parser a -> Parser [a]`. It should apply a parser until it fails.
2. Use many to write an int parser, with type `Parser Int`. It should accept negative and positive integers.
3. Modify your int parser to throw a `NumericOverflow` exception if it detects a numeric overflow while parsing.

Exercise #20

1. Present a helpful GUI error message if the call to xmlNew returns Nothing.
2. Modify the podcatcher to be able to run with either the GUI or the command-line interface from a single code base. Hint: move common code out of PodMain GUI.hs, then have two different Main modules—one for the GUI, and one for the command line.
3. Why does guiFetch combine worker functions instead of calling statusWindow twice?

Exercise #21

1. The Chan type is implemented using `MVars`. Use `MVars` to develop a `BoundedChan` library. Your `newBoundedChan` function should accept an `Int` parameter, limiting the number of unread items that can be present in a `BoundedChan` at once. If this limit is hit, a call to your `writeBoundedChan` function must block until a reader uses readBoundedChan to consume a value.
2. Although we've already mentioned the existence of the strict-concurrency pack- age in the `Hackage` repository, try developing your own, as a wrapper around the built-in `MVar` type. Following classic Haskell practice, make your library type safe, so that users cannot accidentally mix uses of strict and nonstrict MVars.

Exercise #22

1. It can be difficult to determine when to switch from `parSort2` to sort. An alternative approach to the one we outline previously would be to decide based on the length of a sublist. Rewrite `parList2` so that it switches to sort if the list contains more than some number of elements.
2. Measure the performance of the length-based approach and compare it with the depth approach. Which gives better performance results?

Exercise #23

1. Our use of genericLength in easyList will cause our function to loop infinitely if we supply an infinite list. Fix this.
2. Difficult: write a `QuickCheck` property that checks whether the observed false positive rate is close to the requested false positive rate.