# Introduction and Resources

Exercises marked with a star are especially challenging. Whether or not an exercise has a star on it, you should always feel free to ask for help. I have intentionally written more exercises than I think you will finish during class time, so feel free to jump around a lot if you get stuck and/or bored. If you manage to finish everything, tell me and I'll come up with something for you to do.

The nature of this class is such that you will have to spend some time doing Internet research and learning topics on your own. The following are good resources to get you started:

- HaskellWiki (`http://www.haskell.org/haskellwiki/Haskell`). A large, community-maintained Wiki devoted to the Haskell programming language. A good first place to look for information on a specific topic.
- Hoogle (`http://www.haskell.org/hoogle/`). A search engine for the Haskell standard libraries. You can look for any Haskell function either by name or by type. Searching by type is very useful if you find yourself asking the question "Is there a built-in function that does this?"
- Learn You a Haskell for Great Good! (`http://learnyouahaskell.com/`). This is a book, also available online, that aims to be a beginner's guide to Haskell. It covers the topics it covers pretty thoroughly, and it's written in a goofy, easy-going tone. I also have a paper copy of this book if you want to look at it.
- Real World Haskell (`http://book.realworldhaskell.org/`). This book aims to teach the reader how to use Haskell for practical tasks. The online version also lets you see comments from readers.
- HaskellWiki's list of tutorials (`http://www.haskell.org/haskellwiki/Tutorials`) mentions the above two books and also has links to several more.
- The Haskell Cheatsheet (`http://cheatsheet.codeslower.com/`). This is meant to be a "quick reference" for Haskell syntax and context. Good if you remember the general idea of something but have forgotten exactly how you're supposed to type it.
- The Typeclassopedia (`http://www.haskell.org/haskellwiki/Typeclassopedia`). This is a good document to read after you're comfortable with most of the material covered in this problem set. It's a quick but thorough rundown of some of the most important typeclasses in Haskell.
- A Gentle Introduction to Haskell (`http://www.haskell.org/tutorial/`). Despite the title, this tutorial is not at all gentle and is aimed at people who already know a functional programming language. It might be more useful as a resource than as something to learn from.
- The GHC documentation (`http://www.haskell.org/ghc/docs/latest/html/`). Very complete, but far from a tutorial. Good if you already know what you're looking for.
- The Haskell 98 Language Report (`http://www.haskell.org/onlinereport/`). The complete specification of the Haskell programming language.

# 1 Problems That Don't Involve Monads

1. Look up how *list comprehensions* and *ranges* work in Haskell. Use them to write a function that takes a number `n` to the list of all pairs of numbers between `1` and `n` whose sum is a multiple of 4.

2. Use a fold to write the factorial function in one line in a way that doesn't have any explicit recursion. (That is, your function shouldn't call itself, but the fold function you'll use is of course itself recursive.)

3. (a) Write the list of all powers of 3.
   (b) Write the list of all lists consisting of only 1's and 2's, ordered by length.
   (c) Write the list of all prime numbers.

4. (a) Write a function that sorts a list using quick sort. (Hint: Use the `partition` function.)
   (b) Write a function that sorts a list using merge sort.

5. (a) Write a function called `subsets` that takes a list and returns a list of all ordered sublists of it.

```
subsets [1,2] == [[],[1],[2],[1,2]]
subsets "boo" == ["","b","o","bo","o","bo","oo","boo"]
subsets [0,6,2] == [[],[0],[6],[0,6],[2],[0,2],[6,2],[0,6,2]]
```

   The order of the output doesn't have to match the order in these examples. (This is in `Data.List` as `subsequences`.)

   (b) Write a function called `choose` that takes a list and a number and returns all sublists of the list of that length.

```
choose 3 [1..4] == [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
choose 1 "hey" == ["h","e","y"]
choose 2 [66,77] == [[66,77]]
choose 0 "I am the very model of a modern major-general." == [""]
```

   (c) If you didn't already, write a version of `choose` so that `choose k` runs in polynomial time once you've chosen a value of `k`. (In particular, it shouldn't look at the entire output of `subsets`.)

6. Consider the following function definition:

```
f 0 = 1
f 1 = 1
f n = f (n - 1) + f (n - 2)
```

   (a) What does `f` compute?
   (b) How many time will `f` be called to compute `(f 5)`? What about `(f n)` for any `n`?
   (c) Write a version of `f` that uses a list to internally keep track of the values it computes. Make sure it can compute `(f 100)` in a reasonable amount of time. (This technique is called *memoizing* and is a good way to take advantage of Haskell's laziness to minimize repeated computations.)

7. Look up the contents of `Data.List` on the Internet, or just go to `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html` if you can't find it.

   (a) Read the descriptions of the following functions and provide your own implementations.

      i. `reverse`
      ii. `maximum`
      iii. `(!!)`
      iv. `intersperse`
      v. `iterate`
      vi. `unfoldr`
      vii. `break`
      viii. `partition`
      ix. `nub`

   (b) (*) How many of these can be implemented using folds without any explicit recursion?

8. What are the types of the following expressions? (You can check your answers in GHCi after you have a guess.)

   (a) `7`
   (b) `[[], []]`
   (c) `[[], [[]]]`
   (d) `\x y z -> x==7 || y==z`
   (e) `($0)`
   (f) `flip filter [1,2,3]`
   (g) `map (map (<))`
   (h) `map ((map (*)).)`

9. Consider the function `fix` from `Data.Function`, defined like this:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

   (a) What is `fix (0:)`?
   (b) What is `fix ((1:) . map (1+))`?
   (c) Construct a function `f` for which `fix f` can be fully evaluated in finite time.
   (d) (*) Convince yourself that any recursive definition can be rewritten using `fix` without any other explicit recursion. Write the factorial function as `fix f` for some function `f` that has *no* recursion at all.

10. (a) Create a datatype for representing binary trees. Every node should either branch into two subtrees (the *left* and *right* subtrees) or be a leaf. Each node, whether it branches or not, should have a value attached to it, the type of which is arbitrary but the same throughout the binary tree.

    (b) Write a function that...

i. ...outputs the contents of the binary tree as a list. Write three versions: one for preorder, one for inorder, and one for postorder. (Look these up if you don't know them.)

ii. ...outputs the maximum element of a binary tree.

iii. ...adds all the elements of a binary tree.

iv. ...determines whether a binary tree is symmetric, that is, whether flipping the left side about the middle produces the right side.

v. ...takes a tree and an integer `n` and outputs a list of all the nodes at the `n`'th level from the top.

vi. ...outputs the depth of a tree, that is, how far down the deepest node is.

(c) Make your binary tree type an instance of the `Eq` typeclass.

(d) Write a type signature for each function in 10b. Try to make each one as general as possible.

(e) (*) Write a version of `foldr` for binary trees. How many functions from 10b can you implement in terms of your function?

11. This is the definition of the `Monoid` typeclass, defined in `Data.Monoid`:

```haskell
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

It is meant to represent values that can be combined with some binary operation and which have an "identity" with respect to that operation. For example, you could define an instance of `Monoid` for integers like this:

```haskell
instance Monoid Integer where
    mempty = 0
    mappend = (+)
```

This isn't actually the definition that appears in `Data.Monoid`. There are two instances of `Monoid` for numeric types, depending on whether we want the operation in question to be addition or multiplication (look up how `newtype` works if you're not familiar with it):

```haskell
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)


newtype Product a = Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

In order to be a monoid, the `mappend` operation is supposed to be associative (but not necessarily commutative) and `mempty` should be the identity for it, that is, `x `mappend` mempty == mempty `mappend` x == x` for any `x`. Having the `Monoid` typeclass around lets us write functions more generally to work on any monoid — for example, the `mconcat`

function that's part of the typeclass specification above generalizes the built in `sum` and `product` functions among others. We'll see another example in the next problem.

(a) Write the built-in `sum` and `product` functions in terms of monoid functions.

(b) Make lists an instance of `Monoid` under concatenation. (This is actually already part of `Data.Monoid`.)

(c) Look up the `compare` function and the `Monoid` instance for `Ordering`. Use this monoid structure to write a function that compares strings in dictionary order. (The `zipWith` function might be useful. Note that this way of comparing strings is the built-in one.)

(d) If `b` is a `Monoid` and `a` is any type at all, make a `Monoid` instance for `a -> b`.

(e) (*) Look up the functions `sortBy` (in `Data.List`) and `comparing` (in `Data.Ord`) and use the previous part to write a function that sorts a list of strings first alphabetically and then by length.

12. This problem is the first step on our way to writing an interpreter. Our interpreter will run in three steps. The first step, called *parsing*, involves taking the text of the program and turning it into a data structure, called an *abstract syntax tree* or *AST*, which represents the program. The second step involves taking the AST and running the corresponding program.

    We will begin tackling the second part. These are the data structures we will worry about in this problem:

```haskell
data Expression =
    Var String |                            -- a variable
    IntLit Integer |                        -- an integer literal
    BoolLit Bool |                          -- a boolean literal
    UnaryOp String Expression |             -- a unary operation (takes one value)
    BinaryOp String Expression Expression   -- a binary operation (takes two values)
        deriving Show

data Value = IntVal Integer | BoolVal Bool deriving Show
```

    The file `Parse.hs` that I've provided exports a function `parseExpression :: String -> Expression` which might be useful for testing purposes. (If you want to figure out parsing for yourself later on in Problem 22, be sure not to peek!) For example:

```haskell
parseExpression "-(x/2)" == UnaryOp "-" (BinaryOp "/" (Var "x") (IntLit 2))
parseExpression "a and not b" == BinaryOp "and" (Var "a") (UnaryOp "not" (Var "b"))
parseExpression "x+5*y" == BinaryOp "+" (Var "x") (BinaryOp "*" (IntLit 5) (Var "y"))
parseExpression "true or x > y + 5" ==
    BinaryOp "or" (BoolLit True) (BinaryOp ">" (Var "x") (BinaryOp "+" (Var "y") (IntLit 5)))
```

    You should play with the `parseExpression` function for a while if it's not clear.

    Write a function `evalExpr :: [(String, Value)] -> Expression -> Value` which takes an expression to a value. The first parameter is an *environment* which contains a correspondence between variable names and values. Throw an error if you are unable to evaluate the expression. The operators we will use are `"+"`, `"-"`, `"*"`, `"/"`, `"<"`, `">"`, `"=="`, `"and"`, `"or"`, and `"not"`. All are binary except `"not"` and the unary version of `"-"`. (You may find the built-in function `lookup` somewhat useful.)

# 2 Problems That Involve Monads

13. (a) Write `lengths :: IO ()` which, when executed, continually asks the user for input and, every time the user hits enter, prints the length of the line they just typed. For example (every other line is typed by the user):

```
> lengths
hey
3
seven
5
four
4
```

    (b) Generalize `lengths` to a function `repeatedly :: Show a => (String -> a) -> IO ()` so that `repeatedly f` repeatedly asks the user for a line `s` of input and prints `f s`; for example, `lengths` is `repeatedly length`.

    (c) Generalize this to `while :: Show a => (String -> Bool) -> (String -> a) -> IO ()` so that `while p f` acts like `repeatedly f` until `p f` is false:

```
> while (\s -> length s < 2 || s!!1 /= 'b') reverse
hello
"olleh"
kayak
"kayak"
ebb
```

14. Write an `IO Bool` that makes the user play the "guess my number" game. The computer picks a random number between 1 and 100, asks the user for a guess, and tells the user whether its number is higher or lower than the guess. The user should be allowed to guess six times. If the number has been guessed by then, return `True`; otherwise return `False`. (The built-in `randomRIO` might be helpful.)

15. (a) Look up the functions `fmap`, `liftA`, `(<$>)`, and `liftM`. What is the relationship between them?

    (b) What about `liftM2` and `liftA2`? Can you define a version of `liftM2` or `liftA2` for functors? If so, do it; if not, why not?

    (c) Define `liftA2` in terms of `(<$>)` and `(<*>)`.

16. We're going to adjust the interpreter from Problem 12 to throw errors more intelligently. (The code you write in this problem won't be used in future problems about the interpreter.) The type `EvalError` is defined in `Types.hs` as follows:

```
data EvalError =
    UnknownVariable |   -- tried to use a variable that wasn't in the environment
    UnknownOperator |   -- referred to a nonexistent operator
    TypeError           -- tried to apply an operator to the wrong type of input
        deriving Show
```

    Use the fact that `Either EvalError` is a monad to rewrite the `evalExpr` function from Problem 12 to return an `Either EvalError Value`.

17. Look up the contents of `Control.Monad` on the Internet, or just go to `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html` if you can't find it. Read

the descriptions of the following functions and provide your own implementations. Try to figure out what they should do just from the type, the name, and the information on the page.

(a) `mapM` (Try not writing this in terms of `sequence`.)

(b) `sequence`

(c) `forever`

(d) `foldM`

(e) `filterM`

(f) `replicateM`

(g) `join`

18. Do Problem 11 before this one.

(a) The typeclass `MonadPlus` is for monads that are also monoids:

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Make lists an instance of `MonadPlus`. Also make `Maybe a` an instance of `MonadPlus` so that adding two values produces the second value if the first is `Nothing` and otherwise produces the first value. (This is also already defined in the standard libraries.)

(b) Use the `mplus` you just wrote for `Maybe` and the built-in function `find` to write, in 60 characters or less, a function `f :: Num a => [a] -> Maybe a` that takes a list of numbers and returns the first number less than 3 or, if there isn't one, the first number greater than 3.

(c) Look up the `guard` function in `Control.Monad`. What are these functions' types? What do they do?

```
f p xs =
    do x <- xs
       guard $ p x
       return x

g n xs =
    do i <- [1..n]
       map (\x -> (i,x)) xs
```

(d) (*) Write a Sudoku solver. There is code in the file `Sudoku.hs` to get you over the tedious bits.

19. Do Problem 10e before this one. Look up the contents of `Data.Foldable` on the Internet, or just go to `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Foldable.html` if you can't find it. Read the descriptions of the following functions and provide your own implementations.

(a) `traverse_`

(b) `mapM_`

(c) `maximum`

(d) `elem`

20. The `State` monad allows you to perform computations that can read and modify some state. It's defined as a `newtype` wrapper, and it's an instance of `Monad` in the following way:

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
    return a = State $ \s -> (a, s)
    m >>= k  = State $ \s -> let
        (a, s') = runState m s
        in runState (k a) s'
```

A stateful computation is a function that takes an initial state and computes some value and a final state. Then `return a` is the function that keeps the state the same and just outputs `a`, and stateful computations can be composed by feeding the output state from the first one in as the input state to the second one.

Before doing these problems, look up how the `get`, `put`, and `modify` functions work.

(a) Consider the following definitions:

```
x :: State Integer String
x = do modify (+1)
       n <- get
       return $ show n

y = sequence $ replicate 10 x
```

What is the type of `y`? What does `runState y` do?

(b) The `System.Random` module defines a type called `StdGen` for representing random number generators and a function `mkStdGen :: Int -> StdGen` which provides a way of turning a "seed" number into a random number generator in some way.

Because of Haskell's purity, it can be difficult to do things like generate random numbers that require some state to be kept. Look up the built-in functions `random` and `randomR` and use the `State` monad to write a function that takes an integer `n` and a `StdGen` and produces a list of `n` random integers in which the `i`'th is taken from the range `[1..i]`.

21. This problem continues our work on the interpreter which was begun in Problem 12. You should also probably do Problem 20 before this one.

In doing research for the problem about `State`, you may have noticed references to a typeclass called `MonadState`, for example in the types of `get` and `put`:

```
get :: MonadState s m => m s
put :: MonadState s m => s -> m ()
```

Of course, `State s` is an instance of `MonadState`, but with `get` and `put` having these types, they work on more than just `(State s a)`'s. In fact, `get` and `put` are part of the *definition* of `MonadState`.

How do you get new instances of `MonadState`? One way is to use the built-in *state monad transformer*:

```
newtype StateT s m a = StateT {runStateT :: s -> m (a, s)}
```

Monad transformers give ways of combining the effects of monadic computations. For example, `StateT Integer IO String` represents computations that return a `String` but,

along the way, may have I/O effects and can also read from and write to a state of type `Integer`

`StateT` isn't the only monad transformer out there; there's a `ListT`, a `MaybeT`, and many more. You can read more about them in the references at the beginning of this document.

You've already written a function for our interpreter that evaluates expressions; now we'll do the part that runs whole programs. This is the type we'll use:

```
data Statement =
    Assign String Expression |              -- assign a value to a variable
    If Expression Statement Statement |     -- if the expression evaluates to true,
                                              --  execute the first statement, otherwise
                                              --  the second
    While Expression Statement |            -- execute the statement as long as the
                                              --  expression evaluates to true
    Block [Statement] |                     -- execute these statements in order
    Print Expression |                      -- print the value of the expression
    Empty                                   -- do nothing
        deriving Show
```

Before, we used the type `[(String, Value)]` to represent the correspondence between variables and values. Since we're now going to be repeatedly updating it now, it might be easier to represent it using the type `Map` from `Data.Map`. If you want to go this route, look up how `Map`s work, especially the functions `lookup` and `insert`. Otherwise you'll have to write a function that modifies the `[(String, Value)]` environment when a value is assigned to a variable that already exists.

Write a function that takes a `Statement` to a monad that evaluates it. The file `Eval.hs` has a template you can work from. There are helper functions `parseString :: String -> Statement` and `parseFile :: String -> IO Statement` in `Parse.hs` that you can use to generate abstract syntax trees.

22. The final step in writing the interpreter is writing the parser that transforms strings into AST's. To do this, you should work from the template in `ParseTemplate.hs`. We'll be using the Parsec package, which defines a monad specially designed for parsing. Do some research on Parsec package before starting. In particular, learn how to use `buildExpressionParser`, `Text.ParserCombinators.Parsec.Token` (particularly the functions mentioned in the template), and the operator `(<|>)`.

# Language Specification

This is an informal description of the language you'll be writing an interpreter for in Problems 12, 21, and 22.

An *expression* is one of the following things:

- A variable name, which must begin with a letter and contain only letters, numbers, and underscores.
- An integer.
- One of the constants `true` or `false`.
- A unary operator applied to an expression. The unary operators are `-` and `not`.
- A binary operator applied to two expressions. The binary operators are `and`, `or`, `+`, `-`, `*`, `/`, `==`, `<`, and `>`.
- Parentheses surrounding another expression.

There are two *types* an expression might have: integer and boolean. The operators work on types in the way you would expect. It's not possible to tell the type of an expression at compile time because variables can change types as they are assigned new values.

A *statement* is one of the following things:

- An assignment: `v := e;`, where `v` is a variable name and `e` is an expression. This has the effect of evaluating `e` and assigning the resulting value to `v`.
- An `if` statement: `if c then s; else t;`, where `c` is an expression which should evaluate to a boolean and `s` and `t` are statements. If `c` evaluates to `true`, then `s` is executed, otherwise `t` is. The `else t;` can be omitted, and both `s` and `t` are allowed to be blocks.
- A `while` statement: `while c do s;`, where `c` is an expression which should evaluate to a boolean and `s` is a statement. This has the effect of executing `s` as long as `c` keeps evaluating to `true`. As before, `s` can be a block.
- A `print` statement: `print e;`. This prints the value of `e` followed by a newline.
- A block statement: `{s1; s2; ... sn;}`. This executes the enclosed statements in order. Useful for the bodies of `if` and `while` statements.

All statements except for blocks (and therefore `if` and `while` statements which end in blocks) should end in a semicolon.

The following program computes the factorial of 7 and prints it.

```
a := 7;
fact := 1;

i := a;
while i > 0 do
{
    fact := fact * i;
    i := i - 1;
}

print fact;
```