* Chapter 1 – Introduction
  + About Haskell
    - Purely functional programming language
      * Referential transparency – functions called twice with the same inputs are guaranteed to return the same result
    - Lazy – evaluates only when explicitly told to do so
      * Allows programs to function as a series of transformations of input data
    - Statically typed
    - Type inference
  + GHC
    - Takes a Haskell script and compiles it
    - ghci – interactive command
      * :l myfunctions to load myfunctions.hs (must be in same directory)
      * :r to reload the current script
* Chapter 2 – Starting out
  + Arithmetic
    - +, -, \*, / (defaults to floating points)
    - Parentheses have precedence
  + Boolean
    - &&, ||, not
  + Testing equality
    - ==, /=
  + Type inference
    - GHC will complain
    - 5 + 4.0 will work (+ expects numbers and forces 5 to be a decimal)
  + Everything is a function
    - Some operators are infix (in between operands)
    - Most are prefix
    - Functions are called using name and then operators with spaces
      * succ 10
    - Parentheses are used to indicate precedence
    - Prefix functions with two parameters can be made infix
      * 90 `div` 10
    - When writing functions, it doesn’t matter their order as long as they’re all loaded
    - ‘ is a valid symbol to use in name
    - Cannot use capital letters to start the name of a function
  + Intro to lists
    - Homogenous – elements must be the same type
    - Denoted by square brackets
    - Append lists using ++ operator (goes through every element)
    - : is the cons operator (instantaneous)
    - list !! index to get the element at the index
    - Comparison operators can be used on lists (lexicographical order)
    - Common list functions
      * head – first element
      * tail - rest elements
      * last – last element
      * init – first elements
      * length
      * null – checks if empty list
      * reverse
      * take # - takes the first # elements of the list out
      * drop # - drops the first # elements and returns the leftover
      * maximum
      * minimum
      * sum
      * product
      * elem – returns if element exists in list
      * cycle – creates an infinite list (so you have to use take or it will go forever)
      * repeat – repeats an element forever as a list
      * replicate len element – creates a list of length len of only element
    - Ranges [1..20], [‘a’..’z’], [2,4..20]
      * Useful, infinite lists: take 24 [13,26..] (gives the first 24 in the infinite list)
    - List comprehensions
  + Tuples
    - Like lists but for specific sizes
    - Can be heterogeneous
    - Useful function
      * fst – returns the first component of a pair
      * snd – returns the second component
      * zip l1 l2 – takes two lists and puts them in tuples (longer list gets cut off, but then you can use the infinite list)
* Chapter 3 – Types and Typeclasses
  + Haskell infers types
  + :t to display type (short for :type)
  + Can give function type declaration
    - name :: type -> type (-> …)
  + Common types
    - Int – a bounded integer (has a min and max value)
    - Integer – an unbounded integer
    - Float – real floating point with single precision
    - Double – real floating point with double precision
    - Bool – True or False
    - Char – a character
    - String – list of characters
    - Tuples are also their own type (based on size)
  + Type variables
    - Functions like head have a type variable
    - head :: [a] -> a
    - Like generics (more powerful cause they have no type specific behavior)
    - Also called polymorphic functions
  + Typeclasses
    - Similar to interfaces (like Java interfaces but better)
    - If type is part of a typeclass it supports and implements that behavior
    - :t (==)
      * (==) :: (Eq a) => a -> a -> Bool
      * => is a class constraint
      * Basically says it takes two arguments of the same type, a, and return a bool. The type a must belong to the Eq class
    - Examples
      * Eq – supports equality checking (== and /=)
      * Ord – Supports comparison (must also belong to Eq)
      * Show – Supports conversion to string (basically all types except functions)
      * Read – Converts strings to literal values
      * Enum – members are sequentially ordered (they can have ranges and can use the succ and pred functions)
      * Bounded – have upper and lower bounds
      * Num – has the property of being a number
        + Whole numbers are polymorphic constants, until otherwise defined
        + Must also belong to Show and Eq
      * Integral – a Num sub-class that includes only whole numbers
        + fromIntegral – a useful function for converting to floats from ints
      * Floating – a Num sub-class that includes only floating-point numbers
    - Type annotations
      * read “5” :: Int -> 5
* Chapter 4 – Syntax in Functions
  + Pattern matching
    - Functions will pattern match arguments if you define them with different arguments
    - Helps with recursive functions (base cases)
    - Always create a catch all so pattern matching doesn’t fail
    - Helpful example: modifying tuples
    - Pattern matching on lists
      * x:xs – will bind head to x and rest to xs
      * x:y:z:zs – will only match lists with 3 or more elements and will bind accordingly
    - error function makes an error
    - as patterns
      * Kind of like hashing lists into variables?
      * all@(x:xs)
        + all = the whole list
        + x = the first element
        + xs = the rest
      * ++ cannot be used in pattern matching
  + Guards – testing if value has certain properties (like ifs but more readable)
    - After the function name and parameter, with a pipe and a condition
    - If evaluated to true, it’ll use that function body
    - Can be written inline, but doesn’t look good
  + Bindings
    - where
      * Allows for values to be defined and not repeated
      * Defined at the end, exposed to the whole function (including all guards)
      * Syntactic constructs
    - Let
      * Can be defined anywhere
      * Very local to where they are defined
      * Need (most of the time) a complement in to define where that let exists
      * Actual expressions (can be put anywhere)
      * Can be used in list comprehensions (don’t need an in with these)
  + Case expression
    - Case expression of pattern -> result
      * pattern -> result
      * …
* Chapter 5 – Recursion
  + Can use pattern matching, guards and bindings to make readable and quick recursive functions in Haskell
* Chapter 6 – Higher Order Functions
  + Higher order functions = functions that return functions
  + Curried functions – all functions take 1 parameter
  + Function application – an operator that has the highest precedence in Haskell
  + max :: (Ord a) => a -> (a -> a)
    - max takes an a and then returns a function that takes an a and returns an a
  + Calling a function with too few parameters results in a partially applied function
    - Partial applications allow for creating functions on the fly
    - Sectioning – Creating partial infix functions
      * \*\* note: to section with subtract = (subtract 4) not (- 4)
  + Notes about higher order
    - Functions can take functions as parameters
      * When doing this, must include parentheses in the type declaration
    - This is how functional programming languages abstract away common patterns
  + Maps and filters
    - Map takes a function and applies it to every list element
    - Filter takes a predicate and a list and returns those that satisfy the predicate
    - takeWhile returns elements of a list while a given predicate is true
  + Lambdas
    - Long anonymous functions
    - Only want to use them once
    - Use a ‘\’ parameters -> function body
    - Can have multiple parameters
    - Can have pattern matching (only 1 case though, several will fail)
    - Can be used to explicitly denote that a function should be partially applied
  + Folding
    - Always takes a list and a binary function (2 params)
    - Reduces a list to 1 element (the accumulator)
    - foldl – left fold (foldl1 for no base value)
    - foldr – right fold (foldr1 for no base value)
    - scanl and scanr – like fold, but they return a list of intermediate steps
  + Function application ($)
    - Like space, but instead of highest precedence, it has the lowest
    - Reduces parentheses (and keystrokes)
  + Function composition
    - Dot operator allows for composition
    - Can compose multiple functions
    - Allows for point free style
* Chapter 7 – Modules
  + Loading modules
    - Module – a collection of functions, types and classes
    - Program is a main module that will load other modules
    - General, loosely coupled modules can be exported and used again and again
    - Modules can be added to interactive environment using ‘:m + <module name>’
  + Making modules
    - Must be on same level
    - Can be divided into smaller submodules through folders
* Chapter 8 – Making Our Own Types and Typeclasses
  + Intro to data types
    - data keyword
    - name
    - the various values
    - deriving keyword allows for Typeclasses to be applied to the data type
  + Record syntax
    - Giving names to complex data types’ fields
    - Allows the fields to be set in any order, if they’re all there
    - Not good for things like vectors, but good for if you had a Person data type
  + Type parameters
    - Takes types as parameters to produce new types
    - Has parallels to templates in C++
    - Maybe is a good example
      * Nothing
      * Just a
    - Only beneficial when it makes sense (don’t overuse)
    - Related side note: Don’t add type constraints in data declarations
      * You’ll end up specifying them in functions anyways
      * See 3D Vector declaration
  + Derived instances
    - Type can be an instance of a typeclass
      * Int is part of the Eq typeclass
    - If a datatype is going to derive a typeclass, all its fields must support that typeclass as well
  + Type synonyms
    - type String = [Char]
    - Clears up code
    - Allows programmers to give more inherent meaning to the code they write
    - All this allows is for types to be called different things
    - It does not allow that synonym to be used like a data type however
    - Related: maybe vs. either
      * Maybe is for if there is a function that maybe is there or is nothing
      * Either is for if there are two possible conditions
        + See locker example
  + Recursive data structures
    - Data structures can be defined recursively
    - Functions can be defined as infix if they are made of only special characters
      * Number given to define fixity (how tightly they bind)
    - In defining operators, pattern matching should be used based on the constructor
  + Typeclasses 102
    - class keyword for defining a new class
    - Instance keyword allows for deriving a typeclass in a new way
      * Sometimes only parts have to be overridden for the whole thing to work
    - Typeclasses can inherit from other classes by using the => symbol
    - Can apply class restraints to instance declarations
      * See Maybe and Eq
    - :info in interactive compiler can show information about typeclasses
      * functions it defines
      * lists types in the typeclass
  + A yes-no typeclass
    - See code
    - Id function is the identity function
  + Functor typeclass
    - Typeclass for things that can be mapped over
    - Works on types that can act like boxes
      * Lists
      * Maybe
    - Functors have laws associated with them that will be addressed in later Chapters
  + Kinds and some type-foo
    - Kinds = a type of a type
      * :k command to view
      * \* means a concrete type
      * \* -> \*, takes a concrete type and returns another
      * etc
    - These are not usually done by hand, but give a good understanding of the underlying type system in Haskell
* Chapter 9 – Input and Output
  + Basics
    - Hello World -> see helloworld.hs
      * ghc --make target\_file
      * ./target\_output
      * putStrLn prints to the
    - I/O actions are only performed by main
      * Always has the type signature of main :: IO something
    - getLine
      * <- to bind to a name (constructs)
      * IO String
      * Once it binds to the name, it becomes just a string to be referenced or used later
    - In a main do block, let can be used similar to list comprehensions (without an in keyword)
    - runhaskell command allows a program to be run on the fly
    - return keyword
      * Opposite of <-
      * Doesn’t end method/subroutine
      * Returns an IO String
  + Files and Streams
    - Mostly contained in Chapter 9 folder, examples
    - Handle is where a file is located
    - hGetContents is lazy, only reading input as needed
    - Because reading handles is lazy:
      * Lists and files are like streams
      * Default buffering for text files is by lines
      * Default buffering for binary files is by system-defined block size
      * Can be modified by using hSetBuffering
        + Reading larger blocks helps if we don’t want the disk to be accessed a lot
  + Command Line Arguments -> see todo.hs
  + Randomness
    - RandomGen – types that are sources of randomness
    - Random – types that can take on random values
  + Bytestrings
    - Like lists but better for big file inputs
    - Each element is only one byte
    - Handles laziness differently
    - Two flavors
      * Strict – Data.Bytetsring = no laziness at all
      * Lazy – Data.Bytestring.Lazy = Evaluates in chunks
  + Exceptions
    - Mostly for IO situations
    - Maybe and Either can help protect in some situations
    - Head on empty list, divide by 0
* Chapter 10 – Functionally Solving Problems
  + Walks through functional ways of solving two problems
* Chapter 11 – Functors, Applicative Functors, and Monoids
  + Functors redux
    - Functors are things that can be mapped over
    - Partially applied functions are functors
      * What it ultimately boils down to is that it’s just function composition
    - Functor laws
      * Mapping the id function should give back the original functor
      * Mapping a composition of two functions should give them same result as mapping the first function and then mapping the second
  + Applicative Functors
    - Beefed up functors
    - Allow for functions inside of functors to have special application
  + Newtype keyword
    - Used for taking one type and wrapping it in something to present it as another type
    - It’s faster
    - They’re lazier
    - Breaking down type vs newtype vs data
      * Type
        + Used for synonyms
        + New name for existing types
      * Newtype
        + Taking existing types and wrapping them in new types
        + Typically for making them a part of certain Typeclasses
      * Data
        + For making your own data type
  + Monoids
    - An associative binary function and a value that acts as an identity function
    - Multiplying by 1, adding 0, concatenating an empty list
    - Any, all monoids
    - Ordering monoid
    - Maybe monoid
      * Can be a monoid iff its type parameter is a monoid
  + Foldable typeclass – allows data types to be folded over
* Chapter 12 – A fistful of Monads
  + Functors:
    - Things that can be mapped over
    - Lists, maybes, functions, etc.
  + Applicative Functors:
    - Using functors in special contexts
    - Applying across Maybes when the function is inside a maybe
    - Applying a list of functions to a list of values
  + Now Monads!
    - If you have a value in a context, how do you apply a function that takes a normal value and returns a value in a context, so that the function can be applied to a value with a context
    - If we have a fancy value and a function that takes a normal value and returns a fancy value, how do we pass the normal value into the function?
  + Getting our feet wet with maybe
    - Shows how to use monads in the context of a Maybe
  + The Monday type class
    - Has a return function that takes a value and puts it in the minimal default context
      * For Maybe, that would be a Just
    - >>= (bind) – Takes a value with a context and a function that takes a normal value and returns a value in a context, and returns a value in a context
    - The other two functions basic are always default
      * >> is just used as is
      * fail is used to tell programmers why the monad failed in a context
  + Walk the line
    - Monads are really great, especially for preserving failure, because they preserve the context of the value to which a function is applied
  + Do notation
    - Basically, a different way of using monads
    - Used for IO, but can be used elsewhere
  + The list monad
    - List comprehensions are just monads
      * THEY TRICKED US ALL ALONG
  + Monad laws
    - return x >>= f must be the same thing as f x
    - m >>= return must be m
    - Monads should be associative
      * (m >>= f) >>= g must equal m >>= (\x -> f x >>= g)
* Chapter 13 – For a Few Monads More