Chapter 1

Intermissions Excercises (Answers)
1.6

1. b
2. c
3. b

Chapter 1 main points to understand

• Functional programming is based on expressions that incclude variables or constant values, expressions combined with other expressions, and functions.

• Functions have a head and a body and are those expressions that can be applied to arguments and reduced, or evaluated, to a result.

• Variables may be bound in the function declaration, and every time a bound variable shows up in a function, it has the same value.

• All functions take one argument and return one result.

• Functions are a mapping of a set of inputs to a set of outputs. Given the same input, they always return the same result.

Chapter 2

The last case is known as sectioning and allows you to pass around partially applied functions. With commutative functions, such as addition, it makes no difference if you use (+1) or (1+) because the order of the arguments won't change the result. If you use sectioning with a function that is not commutative, the order matters:

```
Prelude> (1/) 2
0.5
Prelude> (/1) 2
2.0
```

Argument vs Parameter

Parameter is the official parameter that a function takes

i.e

f `x` = `x` + 2

`x` is a parameter

Argument is the function input to which the function is applied to. For example if function `f` gets applied to `2`

i.e f `2` = `2` + 2 = `4`

argument is basically the value `2` where `x` is the parameter

Chapter 2

Summary Definitions

1. The terms argument and parameter are often used interchangeably. However, it is worthwhile to understand the distinction. A parameter, or formal parameter, represents a value that will be passed to the function when the function is called. Thus, parameters are usually variables. An argument is an input value the function is applied to. A function's parameter is bound to the value of an argument when the function is applied to that argument. For example, in f x = x + 2 which takes an argument and returns that value added to 2, x is the one parameter of our function. We run the code by applying f to some argument. If the argument we passed to the parameter x were 2, our result would be 4. However, arguments can themselves be variables or be expressions that include variables, thus the distinction is not always clear. When we use "parameter" in this book, it will always be referring to formal parameters, usually in a type signature, but we've taken the liberty of using "argument" somewhat more loosely.
2. An expression is a combination of symbols that conforms to syntactic rules and can be evaluated to some result. In Haskell, an expression is a well-structured combination of constants, variables, and functions. While irreducible constants are technically expressions, we usually refer to those as "values", so we usually mean "reducible expression" when we use the term expression.
3. A value is an expression that cannot be reduced or evaluated any further. 2 * 2 is an expression, but not a value, whereas what it evaluates to, 4, is a

value.

4. A function is a mathematical object whose capabilities are limited to being applied to an argument and returning a result. Functions can be described as a list of ordered pairs of their inputs and the resulting outputs, like a mapping. Given the function f x = x + 2 applied to the argument 2, we would have the ordered pair (2, 4) of its input and output.
5. Infix notation is the style used in arithmetic and logic. Infix means that the operator is placed between the operands or arguments. An example would be the plus sign in an expression like 2 + 2.
6. Operators are functions that are infix by default. In Haskell, operators must use symbols and not alphanumeric characters. 7. Syntactic sugar is syntax within a programming language designed to make expressions easier to write or read.

Chapter 3

Strings

:set prompt "GandalfSays>"
that changes Prelude> to whatever u want, super cool

'a' a is a Character
"a" is a [Char]

The (:) operator, called cons, builds a list:
"c" : "ris"
"cris"

Functions commonly used in Haskell on lists

`head` takes first of list
`tail` removes first and returns the remaining elements
`take n` returns the n number of elements
`drop n` removes the n number of elements

The infix operator, (!!), returns the element that is in the specified position in the list.
Starts with index position `0`

Chapter 4: Basic Datatypes

End definitions
2. A typeclass is a set of operations defined with respect to a polymorphic type. When a type has an instance of a typeclass, values of that type can be used in the standard operations defined for that typeclass. In Haskell, typeclasses are unique pairings of class and concrete instance. This means that if a given type a has an instance of Eq, it has only one instance of Eq.
3. Arity is the number of arguments a function accepts. This notion is a little slippery in Haskell as, due to currying, all functions are 1-arity and we handle accepting multiple arguments by nesting functions.
8. Polymorphism in Haskell means being able to write code in terms of values which may be one of several, or any, type. Polymorphism in Haskell is either parametric or constrained. The identity function, id, is an example of a parametrically polymorphic function:

Chapter 5 Types

Polymorphic type variables give us the ability to implement expressions that can accept arguments and return results of different types without having to write variations on the same expression for each type.

Parametric & constraint polymorphism

Parametric

Constraint (id function) f a -> a

Parametric polymorphism is broader than ad-hoc polymorphism. Parametric polymorphism refers to type variables, or parameters, that are fully polymorphic. When unconstrained by a typeclass, their final, concrete type could be anything. Constrained polymorphism, on the other hand, puts typeclass constraints on the variable, decreasing the number of concrete types it could be, but increasing what you can do with it by defining and bringing into scope a set of operations.

`a Num a => a` Int `

-- The principal type here is the -- parametrically polymorphic 'a'.
-- Given these types

`(Ord a, Num a) => a` Integer

-- The principal type is -- (Ord a, Num a) => a

3. Type variable is a way to refer to an unspecified type or set of types in Haskell type signatures. Type variables ordinarily will be equal to themselves throughout a type signature. Let us consider some examples.
4. A typeclass is a means of expressing faculties or interfaces that multiple datatypes may have in common. This enables us to write code exclusively in terms of those commonalities without repeating yourself for each instance. Just as one may sum values of type Int, Integer, Float, Double, and Rational, we can avoid having different (+), (*), (-), negate, etc. functions for each by unifying them into a single typeclass. Importantly, these can then be used with all types that have a Num instance. Thus, a typeclass provides us a means to write code in terms of those operators and have our functions be compatible with all types that have instances of that typeclass, whether they already exist or are yet to be invented (by you, perhaps)

Chapter 6 - Typeclasses

How to write your own instances?
Link to hackage
ed at http:// hackage.haskell.org/package/base/docs/Data-Eq.html.

The good news is there is something you can do to get more help from GHC on this. If we turn all warnings on with the -Wall flag in our REPL (or in our build configuration), then GHC will let us know when we're not handling all cases:
Prelude> :set -Wall
Prelude> :l code/derivingInstances.hs
[1 of 1] Compiling DerivingInstances

Instance Example with parameterized variables
instance Eq a => Eq (Identity a) where (==) (Identity v) (Identity v') = v == v'

Those details aren't important for understanding this code. Just remember:
• a typeclass defines a set of functions and/or values;
• types have instances of that typeclass;
• the instances specify the ways that type uses the functions of the typeclass.

1. Typeclass inheritance is when a typeclass has a superclass. This is a way of expressing that a typeclass requires another typeclass to be available for a given type before you can write an instance. class Num a => Fractional a where (/) :: a -> a -> a recip :: a -> a fromRational :: Rational -> a Here the typeclass Fractional inherits from Num. We could also say that Num is a superclass of Fractional. The long and short of it is that if you want to write an instance of Fractional for some a, that type a, must already have an instance of Num before you may do so
2. An instance is the definition of how a typeclass should work for a given type. Instances are unique for a given combination of typeclass and type.

Chapter 7 - Pattern Matching

```
isItTwo :: Integer -> Bool
isItTwo _ = False
isItTwo 2 = True
```

Pattern matching works from lft to right and outside to inside.
This is read from top to bottom, therefore `_ `
will always get executed and the rest of the patterns wqill not be matched

Try to order your patterns from most specific to least specific, particularly as it concerns the use of _ to unconditionally match any value. Unless you get fancy, you should be able to trust GHC's pattern match overlap warning and should triple-check your code when it complains

7.12 Chapter Definitions

1. Binding or bound is a common word used to indicate connection, linkage, or association between two objects. In Haskell we'll use it to talk about what value a variable has, e.g., a parameter variable is bound to an argument value, meaning the value is passed into the parameter as input and each occurrence of that named parameter will have the same value.

Bindings as a plurality will usually refer to a collection of variables and functions which can be referenced by name. **:: -> blah :: Int ; blah = 10 -> Here the variable blah is bound to the value 10.**

2. An anonymous function is a function which is not bound to an identifier and is instead passed as an argument to another function and/or used to construct another function. See the following examples. \x -> x -- anonymous version of id id x = x -- not anonymous, it's bound to 'id'
3. . Currying is the process of transforming a function that takes multiple arguments into a series of functions which each take one argument and return one result. This is accomplished through the nesting. In Haskell, all functions are curried by default. You don't need to do anything special yourself.
4. **Higher-order** functions are functions which themselves take functions as arguments or return functions as results. Due to currying, technically any function that appears to take more than one argument is higher order in Haskell.
5. **Function Composition** -> -- or (.) :: (b -> c) -> ((a -> b) -> (a -> c)) -- can be implemented as comp :: (b -> c) -> ((a -> b) -> (a -> c)) comp f g x = f (g x) The function g is applied to x, f is applied to the result of g x.
6. **Pointfree** is programming tacitly, or without mentioning arguments by name. This tends to look like "plumby" code where you're routing data around implicitly or leaving off unnecessary arguments thanks to currying. The "point" referred to in the term pointfree is an argument.

Chapter 8 - Recursion

Type
Base case
Recursive case

Chapter 9 - Lists

LIST COMPREHENSIONS NOTES

We can also write list comprehensions that have multiple generators. One thing to note is that the rightmost generator will be exhausted first, then the second rightmost, and so on. For example, let's say you wanted to make a list of x to the y power, instead of squaring all of them as we did above. Separate

the two inputs with a comma as below: Prelude> [x^y | x <- [1..5], y <- [2, 3]]
[1,1,4,8,9,27,16,64,25,125]

zip proceeds until the shortest list ends.
Prelude> zip ['a'] [1..10000000000000000] [('a',1)]
Prelude> zip [1..100] ['a'..'c'] [(1,'a'),(2,'b'),(3,'c')]

zip unzip zipwith
Remember zip combines lists, but basically returns a list of tuples
LIST: OF TUPLES AND EACH TUPLE (indexed1ItemList1, indexed1ItemList2)
i.e
zip [1,2] [3,4]
returns -> [(1,3), (2,4)]

zipWith however does nor return list of tuples, returns lists combined based on the function provided
i.e
zipWith (+) [1,2,3] [1,2,5]

Products & Sum types definitions

1. In type theory, a **product type** is a type made of a set of types compounded over each other. In Haskell we represent products using tuples or data constructors with more than one argument. The “compounding” is from each type argument to the data constructor representing a value that coexists with all the other values simultaneously. Products of types represent a conjunction, “and,” of those types. If you have a product of Bool and Int, your terms will each contain a Bool and Int value.
2. In type theory, a **sum type** of two types is a type whose terms are terms in either type, but not simultaneously. In Haskell sum types are represented using the pipe, |, in a datatype definition. Sums of types represent a disjunction, “or,” of those types. If you have a sum of Bool and Int, your terms will be either a Bool value or an Int value.
3. Cons is ordinarily used as a verb to signify that a list value has been created by cons’ing a value onto the head of another list value. In Haskell, (:) is the cons operator for the list type. It is a data constructor defined in

the list datatype:
(:) :: a -> [a] -> [a]

Reasoning
1 : 2 : 3 : []
3 to empty list []
2 to [3]
1 to [2,3]

Chapter 10 - Folding lists

Catamorphisms are a means of deconstructing data. If the spine of a list is the structure of a list, then a fold is what can reduce that structure.1

-- Remember how map worked?
map :: (a -> b) -> [a] -> [b]
map (+1) 1 : 2 : 3 : [] (+1) 1 : (+1) 2 : (+1) 3 : []
Given the list foldr (+) 0 (1 : 2 : 3 : []) 1 + (2 + (3 + 0))

- Where map applies a function to each member of a list and returns a list, a fold replaces the cons constructors with the function and reduces the list.

Right fold foldr evaluation

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs =
case xs of
[] -> z
(x:xs) -> f x (foldr f z xs)

So one way to think about the way Haskell evaluates is that it's like a text rewriting system. Our expression has thus far rewritten itself from:
foldr (+) 0 [1, 2, 3]
Into: (+) 1 ((+) 2 ((+) 3 0))

Use this in REPL to see how the evaluation works.
xs = map show [1..5]
y = foldr (\x y -> concat ["(",x,"+",y,")"]) "0" xs

take is **nonstrict** like everything else you've seen so far, and in this case, it only returns as much list as you ask for.
Basically strict should mean that we evaluate ALL VALUES AS WELL.
for example
let xs = [1,2,3,4] ++ undefined
length (take 4 xs) -> this will work
length xs will not work.

Difference in foldl reasoning evaluation

Prelude> let conc = concat
Prelude> let f x y = conc ["(",x,"+",y,")"]
Prelude> foldl f "0" (map show [1..5]) "(((((0+1)+2)+3)+4)+5)"

We can see from this that foldl begins its reduction process by adding the acc (accumulator) value to the head of the list, whereas foldr had added it to the final element of the list first.

How to think of evaluation

Associativity and folding Next we'll take a closer look at some of the effects of the associativity of foldl. As we've said, both folds traverse the spine in the same direction. What's different is the associativity of the evaluation. The fundamental way to think about evaluation in Haskell is as SUBSTITUTION. When we use a right fold on a list with the function f and start value z, we're, in a sense, replacing the cons constructors with our folding function and the empty list constructor with our start value

[1..3] == 1 : 2 : 3 : []
foldr f z [1, 2, 3]
1 `f` (foldr f z [2, 3])
1 `f` (2 `f` (foldr f z [3]))
1 `f` (2 `f` (3 `f` (foldr f z [])))
1 `f` (2 `f` (3 `f` z))

Furthermore, lazy evaluation lets our functions, rather than the ambient semantics of the language, dictate what order things get evaluated in. Because of this, the parentheses are real. In the above, the 3 `f` z pairing gets

evaluated first because it's in the innermost parentheses. Right folds have to traverse the list outside-in, but the folding itself starts from the end of the list

foldr :: (a -> b -> b) -> b -> [a] -> b -- [1] [2] [3] foldl :: (b -> a -> b) -> b -> [a] -> b -- [4] [5] [6] 1. The parameter of type a represents one of the list element arguments the folding function of foldr is applied to. 2. The parameter of type b will either be the start value or the result of the fold accumulated so far, depending on how far you are into the fold. 3. The final result of having combined the list element and the start value or fold so far to compute the fold. 4. The start value or fold accumulated so far is the first argument to foldl's folding function. 5. The list element is the second argument to foldl's folding function

Discord user recommendation

Data.Bool.bool gatekeeper — Today at 13:12

`foldr` for lazily generating a list from an existing one, or for calculations that don't require knowing the whole list (e.g. `and`) `foldl'` for calculating summary values that depend on knowing all elements (like `sum`) `foldl` basically never

1. By the 'evaluation process' they mean the difference between `1 `f` (2 `f` (3 `f` []))` (for `foldr`) and `(([] `f` 1) `f` 2) `f` 3` (for `foldl`)

2. *[_12:58]_*

   One evaluates from the right, the other from the left

foldl (-) 0 [1, 2, 3]
= foldl (-) 0 [1, 2] - 3
= (foldl (-) 0 [1] - 2) - 3
= ((foldl (-) 0 [] - 1) - 2) - 3
= ((0 - 1) - 2) - 3
= (-6)

foldr (-) 0 [1, 2, 3]
= 1 - foldr (-) 0 [2, 3]
= 1 - (2 - foldr (-) 0 [3])
= 1 - (2 - (3 - foldr (-) 0 []))

= 1 - (2 - (3 - 0))
= 2

foldr

1. The rest of the fold (recursive invocation of foldr) is an argument to the folding function you passed to foldr. It doesn't directly self-call as a tail-call like foldl. You could think of it as alternating between applications of foldr and your folding function f. The next invocation of foldr is conditional on f having asked for more of the results of having folded the list. That is: foldr :: (a -> b -> b) -> b -> [a] -> b -- ^
2. That b we're pointing at in (a -> b -> b) is the rest of the fold. Evaluating that evaluates the next application of foldr.
3. Associates to the right.
4. Works with infinite lists. We know this because: Prelude> foldr const 0 [1..] 1
5. Is a good default choice whenever you want to transform data structures, be they finite or infinite.

Chapter 11 - Algebraic Datatypes

4. Data constructor. In this case, a data constructor that takes no arguments and so is called a nullary constructor. This is one of the possible values of this type that can show up in term-level code.
5. 5. The pipe denotes a sum type which indicates a logical disjunction (colloquially, or) in what values can have that type. (|)

Although the term constructor is often used to describe all type constructors and data constructors, we can make a distinction between constants and constructors. Type and data constructors that take no arguments are constants. They can only store a fixed type and amount of data.

Distinguish
Type-level (i.e Bool)
Term-level (i.e True or False)

data Bool = True | False

Kinds

Are basically types one level up
U can query them with :k
:k Bool
Prelude> Bool ::
Means concrete and represented with
Type constant is a type constructor with zero arguments
data PubType = PubData
PubType takes 0 arguments so it is called as type constant
PubData is also called as constant value, cause it takes 0 arguments

```
data HuskyType a = HuskyData
```

4. HuskyData is the data constructor for HuskyType. Note that the type variable argument a does not occur as an argument to HuskyData or anywhere else after the =. That means our type argument a is phantom, or, "has no witness." We will elaborate on this later. Here HuskyData is a constant value, like PugData.

Important note

As we've said, types are static and resolve at compile time. Types are known before runtime, whether through explicit declaration or type inference, and that's what makes them static types. Information about types does not persist through to runtime. Data are what we're working with at runtime.

Data constructors are basically generated by the DECLARATION itself